Model-Driven Development of QoS-Enabled Distributed Applications

vorgelegt von
Diplom-Informatiker
Torben Weis
aus Berlin

von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor Ingenieur
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Radu Popescu-Zelletin
Berichter:	Prof. Dr. Kurt Geihs
Berichter:	Prof. Dr. Jean-Marc Jézéquel

Tag der wissenschaftlichen Aussprache: 19.5.2004

Berlin 2004
D 83

Abstract

The term Quality of Service (QoS) describes how well an application or service performs. Hence, QoS can describe properties such as response time, availability, the level of encryption, throughput etc. QoS is important for multi-media (i.e. video and audio applications) as well as for enterprise applications. The presented work focuses on QoS-aware enterprise applications. Enterprise applications are concerned with business transactions such as ordering a product or transferring money between two accounts. QoS properties of enterprise applications determine, for example, how many transactions the system can handle in a certain time frame, how many concurrent users it can serve, or how available the system will be.

Middleware tools and libraries shield the developer almost completely from the realization of distribution. In an ideal world, QoS-enabled middleware could do the same for QoS, i.e. the developer does not have to care how QoS is realized. However, the design of an application is highly dependent on its QoS properties. This means that QoS must be tackled at the design phase and this is not possible with an approach solely based on QoS-enabled middleware. This thesis shows how a model-driven approach can overcome this limitation. The development starts with a platform independent model (PIM), which describes the behavior and QoS properties of an application. This model is automatically transformed by a model transformation into a platform specific model (PSM). The PSM describes the design for a concrete implementation on a specific platform, i.e. operating system, programming language, and middleware. Due to the automatic model transformation, tools can influence the platform-specific design and take care of the PIM QoS properties.

To realize this scenario, two problems had to be solved: the modeling of QoS properties in the PIM and the model transformation. Current approaches to model transformation require in depth knowledge of the modeling language and its internals. To simplify the construction of transformers, the visual rule-based transformation language Kafka and associated tools based on graph transformation theory have been developed throughout this thesis. Kafka simplifies the implementation of transformers because it builds on the notation of the PIM and the PSM. Hence, it shields the developers of model transformers from the internals of the modeling languages, i.e. the meta models. To create PIMs of QoS-aware applications, a modeling language called PIQML has been developed which is based on UML 2.0 components, hierarchical message sequence charts, and a novel meta model extension for QoS contracts. The thesis shows how PIQML models can be mapped to several target platforms, i.e. programming languages and middleware products. Based on PIQML and Kafka, the CASE tool Kase has been developed which integrates the editing of PIQML models (i.e. PIMs), UML models (i.e. PSMs), and Kafka transformations. The result is an integrated tool chain for the model-driven development of QoS-aware distributed applications.

Zusammenfassung

Der Ausdruck Quality of Service (QoS) beschreibt die Güte eines erbrachten Dienstes. QoS beschreibt Eigenschaften wie Antwortzeit, Verfügbarkeit, den Grad der Nachrichten-Verschlüsselung oder den Durchsatz. QoS ist sowohl für Multi-Media Anwendungen (Audio- und Video-Übertragung) als auch für Geschäftsanwendungen wichtig. Die vorgelegte Arbeit konzentriert sich dabei auf das Feld der Geschäftsanwendungen. Solche Anwendungen arbeiten mit Transaktionen wie etwa dem Bestellen eines Produkts oder der Überweisung von Geld zwischen Konten. QoS Eigenschaften solcher Anwendungen sind beispielsweise die Anzahl an Transaktionen während eines Zeitintervalls, die maximale Anzahl gleichzeitig mit dem System arbeitender Nutzer, oder die zu erwartende Verfügbarkeit. Durch Middleware kann die Realisierung von Verteilung vor dem Programmierer verborgen werden. In einer idealen Welt würde Middleware dasselbe für QoS leisten. Das heißt, der Entwickler spezifiziert nur die gewünschte Dienstgüte und kümmert sich nicht darum, wie sie realisiert wird. Das ist so aber nicht möglich, denn die zu erbringende Dienstgüte hat massive Auswirkungen auf das Design einer Applikation. Daher muss QoS schon in der Design-Phase angegangen werden, was mit einer rein Middleware-gestützten Herangehensweise nicht möglich ist. Die vorgelegte Arbeit zeigt, wie dieses Problem durch einen modellgetriebenen Ansatz behoben werden kann. Die Entwicklung beginnt dann mit einem Plattform-unabhängigen Modell (PIM), welches das Verhalten und die QoS Eigenschaften der Applikation beschreibt. Dieses Modell wird automatisch durch eine Modell-Transformation in ein Plattform-spezifisches Modell (PSM) umgewandelt. Das PSM beschreibt das Design einer Implementierung für eine spezifische Plattform und berücksichtigt dabei die im PIM beschriebenen QoS-Eigenschaften.

Um dieses Szenario in der Praxis anwenden zu können, müssen zwei Probleme gelöst werden: das Modellieren von QoS Eigenschaften und die Modell-Transformation. Gegenwärtig erfordert das Erstellen einer Modell-Transformation detailliertes Wissen über die Modellierungssprache und ihre Interna. Um das zu vereinfachen, wurde im Rahmen dieser Arbeit die visuelle Regelbasierte Transformationssprache Kafka entwickelt. Kafka und die zugehörigen Werkzeuge bauen auf Graphtransformations-Theorie auf. Kafka vereinfacht das Erstellen von Transformationen weil es auf der Notation des PIM und PSM aufbaut. Daher muss der Entwickler nichts von den Interna (etwa dem Meta-Modell) verstehen. Um PIMs von QoS-fähigen Applikationen zu erstellen, wurde die Modellierungssprache PIQML entwickelt, welche auf UML 2.0 Komponenten, Hierarchischen Nachrichten Sequenz Diagrammen (HMSCs) und einer neuen Meta-Modell Erweiterung für QoS Verträge basiert. Die vorgelegte Arbeit zeigt, wie PIQML Modelle auf verschiedene Zielplattformen (Programmiersprachen und Middleware Produkte) abgebildet werden können. Basierend auf PIQML und Kafka wurde das Werkzeug Kase entwickelt, welches das Editieren von PIQML Modellen (PIMs), UML Modellen (PSMs) und Kafka Transformationen integriert. Das Ergebnis ist eine integrierte Werkzeug-Kette für das modellgetriebene Entwickeln QoS-fähiger verteilter Applikationen.

Acknowledgement

First of all, I want to thank Prof. Dr. Kurt Geihs for his excellent support of my research and for providing such a creative environment to work in. Special thanks go to Prof. Dr. Jean-Marc Jézéquel for taking over the part of the second reviewer.

Furthermore, I want to thank all colleagues and friends who supported me during the last three years. Especially Andreas Ulbrich did an invaluable job. Without him Kase would still lack Python support and writing papers would have been only half the fun it used to be. Andreas always had time for a discussion about technical issues, movies, or cooking even long after midnight. Klaus Herrmann listened to all my moaning and always had a good scientific or personal advice at hand. Christian Becker got me started on the scientific business and was always available when I needed some help. I want to thank Prof. Dr. Jean-Marc Jézéquel, Dr. Noel Plouzeau, and Karine Odent for their contribution to the QCCS project and for our creative internal workshop in Rennes. Further credits go to Tom Ritter and Marc Born for their work on Qedo and their contribution to the MWQoS I+II projects. Alexander Brändle from Microsoft Research provided Andreas and me with software and tech-toys and rose my funding for the last months of my PhD work.

Special thanks go to my girlfriend Claudia. She had to suffer from weekends and nights of coding. Nevertheless, she always encouraged me in my work and reminded me of the fact that there is a whole world outside a computer window. Finally, I want to thank my parents for more things than I could enumerate on a single page. My father provided me with my first home computer (a Commodore VC 20) at a time when almost nobody knew what this is all about and my mother showed me that cooking can be at least as much fun as eating. Thank you.

Contents

Part I

Introduction

Chapter 1

Motivation

Modern software technology faces a couple of major challenges, most notably *reuse* of existing solutions, *complexity* of software systems, and *distribution*. The problem of sending data via a network from one machine to another has already been addressed by middleware products. However, that alone is not enough. Sophisticated distributed applications need to know how well the communication works. For example, availability, security, delay, and throughput of a communication link are very important. These properties of a service are subsumed under the term Quality of Service (QoS). QoS determines *how well* a service performs. In contrast, functionality determines *what* a service does. QoS and functionality are orthogonal to each other. Protocols and algorithms for the realization of QoS properties have been investigated by research groups. However, many applications fail to support QoS. Neither do they gracefully handle the failure of a remote service nor do they adapt if bandwidth changes. Handling QoS is – in theory – a well-studied problem. The lack of QoS support in practice originates either from the high complexity and associated costs of building QoS-aware applications or from companies not being interested in QoS.

This leads to another challenge, the *complexity* of modern software systems. According to [Whe01, Sch00b] the Windows operating system, for example, is built from 40 million lines of code (LOC). Red Hat Linux 7.1 sums up to 30 million LOC. The NASA Space Shuttle flight control is estimated with 2 million LOC. Even the tools developed throughout this thesis sum up to 150.000 LOC. This complexity must be managed. Usually, a single developer has no overview of the complete code base. This gave raise to software modeling. Models make information about a software system explicit that is only implicitly given by the source code, for example, the structure of a software system. However, models can be more than pure documentation. Modern modeling tools generate an entire executable application from an executable model [Ken03].

The next challenge is that of *software reuse*. Building software is a very expensive and time consuming process. Therefore, existing solutions should be reused to save time and money and to avoid the repetition of previous mistakes.

QoS-aware software development faces all three challenges:

- QoS originates in the domain of distributed systems and networks.
- The development of QoS-aware applications can be quite complex and requires expert knowledge.
- Existing solutions for certain QoS properties should be reusable.

To cope with these challenges, this thesis focuses on the *model-driven development of QoS-aware distributed applications*. A *model* is an abstraction of a complex system. Hence, modeling can be used to handle complexity. It does so by suppressing details that are not relevant and by making information explicit that is otherwise only implicitly given. The concept of modeling has a long-lasting tradition in natural sciences and gains importance in the field of computer science. Models have been used to describe the constitution of atoms and molecules [Boh13] and the scattering of particles by matter [Rut11].

In computer science, models are used to gain a better understanding of applications and their implementation and to foster the creative process of constructing software. It is important to differentiate between a model of an application and a model of its implementation. The model of an application is expressed in terms of its *application domain*. For enterprise applications this includes terms such as client, product, offer, price, transactions, and QoS. A model of the implementation is expressed in terms of the platform used for implementation, i.e. programming language, middleware, operating system, and GUI framework. This is the *implementation domain*. Ideally, a developer describes the application in the application domain and tools automatically generate an implementation.

This thesis shows how an automatic mapping between both domains can be achieved via model transformation. This means, the application model is automatically transformed into an implementation model. Especially QoS-enabled applications can benefit from this approach. Specifying QoS properties is much easier than implementing them. Therefore, the knowledge about how to realize a certain QoS property is moved into the model transformer. The use of such a QoS-aware model transformation can significantly facilitate the development of QoS-enabled applications.

The following two sections further motivate the model-centric approach taken by this thesis. The first section describes the limitations of a solely middleware-based approach. The second section shows how these limitations can be overcome on the modeling level.

1.1 Limitations of QoS-enabled Middleware

Middleware platforms such as CORBA [OMG02a], .NET Remoting [ECM01b, Ram02], or RMI [Gro01] are industrial strength technologies to cope with distribution. These middleware products do not treat QoS as a first class entity. Therefore, it is inevitable to extend the middleware with *QoS specification* and *QoS provisioning*. QoS specification is usually implemented via a QoS specification language – such as QML [FK98a] – or an extension of an interface definition language, for example QIDL [BG98]. QoS provisioning requires runtime support via QoS mechanisms, i.e. resources allocation, message compression, monitoring, etc. However, middleware hides almost all aspects of distribution such as the marshaling of data, queuing of requests, and network protocols. The middleware must be extended to perform bandwidth reservation, encrypted communication, or authentication. Some QoS-aware middleware products feature built-in support for special QoS properties, for example, availability or real time. Others extended their platform with interceptors [Sch00a], pluggable protocols [OMG02a], or customizable proxies [Ram02] to allow arbitrary QoS mechanisms to be hooked into the middleware.

Work on QoS-enabled middleware platforms has shown that QoS is inherently cross cutting [BG98, HBG$^+$01, HBG$^+$99, PLS$^+$00a] in the sense of aspect-oriented programming (AOP) [KLM$^+$97]. This means the QoS-related code cannot be encapsulated in object-oriented systems. QoS can affect the entire message path ranging from the physical layer up to the application layer. Therefore, the use of AOP to encapsulate QoS-related code has been applied to multiple CORBA-based middleware platforms [Bec01, HBG$^+$01, ZBS97a]. Such middleware platforms succeeded in separating QoS-related aspects for all layers except for the application layer.

The design of an application is highly dependent on the QoS mechanisms used. For example, the design of a component that is subject to replication differs from one that cannot be replicated. Either the component is stateless, or it stores its state in a central database, or it uses some protocol to synchronize the replicas. From the design perspective, this is a major difference. Furthermore, QoS mechanisms may need information about the way a component is used, hence, information about the business logic. Typically, only the interfaces of a component are available to the middleware and its tools. In this case the QoS mechanisms cannot know in which sequence the methods of the interface will be invoked. They cannot know how many method calls a client will make to accomplish a certain task. This complicates the planning and resource allocation. Especially the billing of services is problematic. The customer is not interested in purchasing access to some interfaces. He wants to perform certain tasks such as searching an article in an online library or transferring money from his bank account. Handling the billing of such products requires knowledge of the business logic. However, middleware platforms are only concerned about interfaces and not about the way the interface is used to accomplish a task.

Setting up a QoS-aware application requires special considerations during the deployment. The best load balancing does not help if the network is too slow or if the machines are too weak. Fault tolerant software systems require fault tolerant hardware to reach a very high level of availability. Middleware platforms cannot influence the deployment. Middleware-related tools generate source code – long before deployment is done – and they support the application at runtime when the deployment is already finished.

Designing and planning a QoS-aware application cannot be reduced to the independent design of single components. However, that is what middleware platforms suggest developers to do. Developers start by defining interfaces via some interface definition language (IDL). This IDL may be enriched with QoS-related specifications. Then, they generate skeletons and stubs and add the business logic manually in the source code editor. This approach does not yield a design document that specifies which component will interact with which other components, yet such a design document is very important. It shows the coherences in the system and it shows how the QoS of one component may affect the QoS delivered by other components and the application in general. This discussion yields four shortcomings of middleware-centric QoS management:

- limited influence on design,
- the business logic is not available to the middleware,
- deployment is not considered,
- coherence of the application's components is not visible.

This thesis shows how the aforementioned problems can be solved using a model-driven approach.

1.2 Advantages of Modeling

The first mentioned shortcoming of QoS-enabled middleware can immediately be solved at the modeling level. Models capture the design of an application, respectively a design of its implementation. Tools that work with models can inspect and alter the design. In contrast, the code generators (IDL compilers) of middleware platforms can only create skeletons that have to be filled out by the application developer.

A model can be much more than a definition of application structure in terms of classes, associations, and interfaces. For example, the Unified Modeling Language (UML) [OMG03i] or formal models such as Petri Nets [Pet81] can make the *behavior* of a software system explicit. The UML provides a rich set of diagram types for this purpose: sequence diagrams, state charts, activity diagrams, and collaboration diagrams. These diagram types can be used to model the business logic. If QoS is tackled already in the design phase, model-based tools can inspect these diagrams to get information about the business logic. Thus, modeling can tackle the second shortcoming of a solely middleware-based approach.

The UML features a special diagram type for the deployment of a system. Tools can inspect the deployment of components and their QoS properties. If the developers did, for example, not deploy enough replicas or if the network connection of the load balancer is too weak, a tool can detect this and raise an error [WBGP01]. Hence, the third mentioned shortcoming can be eliminated at the modeling level.

Finally, models are well suited to model the coherence of components. For example, UML or EDOC [OMG02f] provide a diagram type dedicated to components and their interaction. Such component diagrams show how components are interconnected and which interfaces they use for their communication. This information is not available to IDL compilers as in CORBA or reflective middleware platforms such as .NET Remoting.

Thus, the shortcomings of middleware platforms can be conceptually solved at the modeling level. To turn this conceptual advantage into practical improvements, developers need special tools that work with models. Therefore, a complete tool chain for model-driven development has been created during the work on this thesis.

Chapter 2

Focus and Contribution

This thesis discusses model-driven development of QoS-enabled distributed applications. Development of large-scale software systems is, ideally, guided by a *methodology*. A methodology "consists [...] of both a modeling language and a process" [Fow97]. "The modeling language is the (mainly graphical) notation" [Fow97] used to express design. The "process defines *who* is doing *what when* and *how* to reach a certain goal" [JBR99]. Several object-oriented and component-oriented methodologies have been developed over the years, among them Catalysis [DW98] and the Rational Unified Process (RUP) [JBR99]. For most of them the UML is the modeling language of choice. However, none of them has addressed the special problems of QoS-enabled distributed systems.

According to [JBR99] a process has to take technologies, tools, people, and organizational patterns into account. It is not the intention of this thesis to create a new process completely from scratch. Instead, the thesis shows how techniques such as modeling, model transformations, middleware, and a specialized tool chain can be used in an "iterative and incremental process" [JBR99]. Hence, process issues such as organizational patterns, formal reviews, building of teams, etc. are not touched by the presented work.

The contribution of this thesis can be divided into (1) the extension of a modeling language and (2) the integration of model transformation and supporting tools in an iterative and incremental process. These two aspects are briefly introduced in the following two sections.

2.1 Modeling Language

Current modeling languages for distributed applications do not feature concepts for QoS. Therefore, this thesis presents a platform independent modeling language (PIQML) for modeling QoS-enabled applications. PIQML builds on UML 1.4 and a subset of UML 2.0. PIQML is component-based, i.e. the application is decomposed into components which are connected via ports.

PIQML adds QoS support via QoS contracts. A QoS contract describes a *measurable quality level* or expresses a constraint on the desired quality level. These contracts can be attached to component ports. Hence, PIQML can express the quality level that a component offers and the quality level that it expects from its environment. In contrast to specialized UML real-time extensions [OMG03g, Dou99], PIQML features concepts for the modeling of different QoS categories such as performance, security, availability, throughput etc.

PIQML introduces the *task* concept. Designers can model a task that consists of a series of method invocations between components. A task can be enriched with QoS contracts. The advantage of this approach is that PIQML can, for example, express the worst case execution time of an entire task. In contrast, other QoS specification languages [BG99, FK98a, Aag01, RZ03, OMG03g] can only express the worst case execution time of single method invocations. Hence, PIQML is well suited for the modeling of distributed enterprise applications. In such enterprise applications single method calls are not as important as in real-time scenarios. Instead, designers are interested in the number of *transactions* a system can process in a given time interval. Tasks can be used to model the activities of such a transaction and to specify the associated quality level.

2.2 Process

Modern processes such as [JBR99] and [DW98] usually build on a common workflow: requirements, analysis, design, implementation, deployment, and testing. This thesis focuses on design, implementation, and deployment. Requirements can be handled via use cases [OMG03i, JBR99]. Integrating testing in model-driven development has become subject to scientific work lately.

Figure 2.1 shows the development process for the three phases design, implementation and deployment as proposed by this thesis. The boxes denote models or source code, hence, physical artifacts. The arrows indicate transitions from one artifact to another. Such transitions are either automated or require manual work. Code generation is not discussed in depth by this thesis since this feature has already found its way into commercially available software. The transition between models is the major challenge since model transformation is still a subject of intensive research (see chapter 4).

The development starts with a very high-level model. It decomposes the application into components. Each component describes how it is to be used and how it behaves. This includes its interfaces, its invocation protocols, and its QoS properties. Hence, the model describes the business logic from a very high-level point of view. Finally, the components are connected to form the system.

The following steps are the most complicated ones. First, the model is automatically transformed into a more low-level one. The low-level model is considerably closer to the source code. It can be treated as a model of the implementation. However, the generated model is – so far – QoS-agnostic. This means that the QoS properties, which are part of the high-level model, have been ignored.

The second transformation step searches for QoS properties in the high-level model and selects a set of QoS aspects. The term *aspect* is borrowed from AOP. An aspect groups *cross-cutting* information, i.e. this information cannot be encapsulated by means of object-oriented constructs such as classes or interfaces. A *QoS aspect* specifies how a QoS-agnostic low-level model has to be modified and extended to realize QoS. QoS aspects are reusable. They depend only on the modeling language used for the high-level model and the target platform of the low-level model. They do not depend on a concrete high-level model.

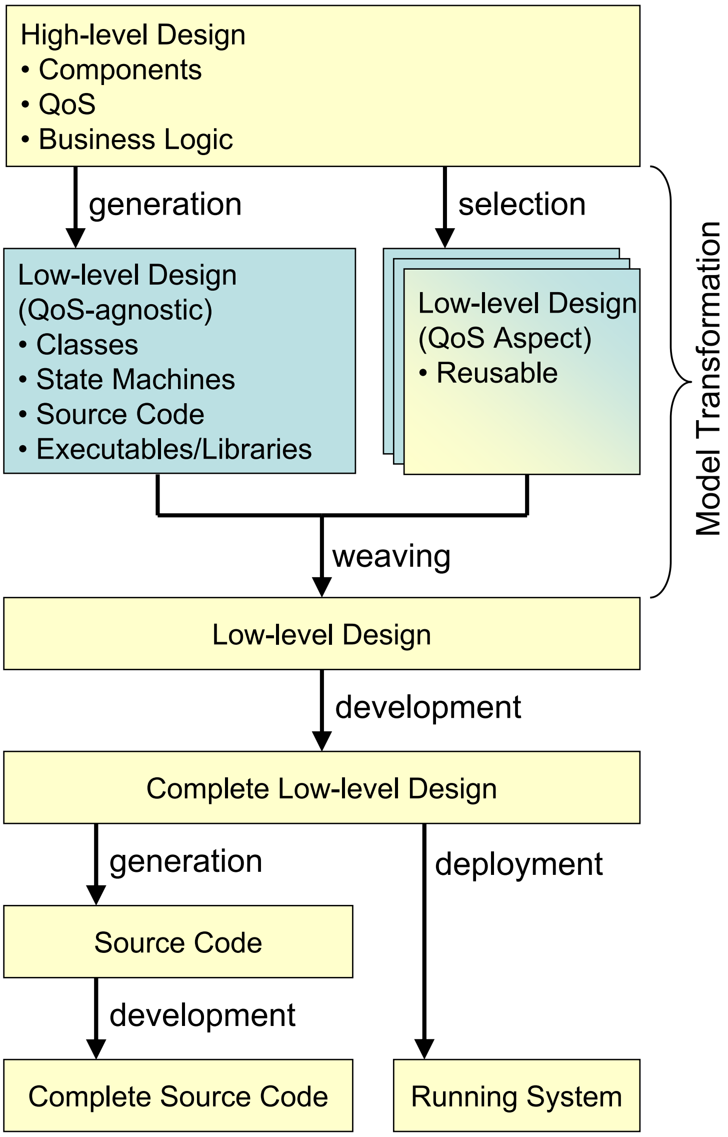


Figure 2.1: Methodology overview

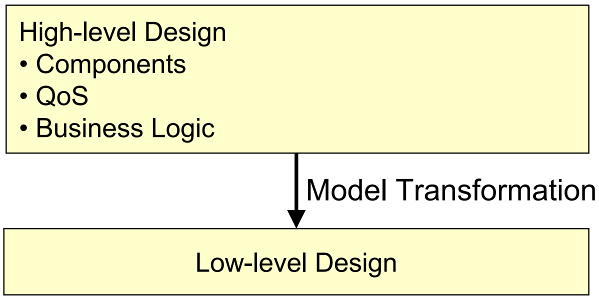


Figure 2.2: Methodology from the viewpoint of a level-one developer

The third step *weaves* [KLM+97] the QoS aspects with the low-level model. The result is a low-level model of the system that takes QoS into account. Part II of this thesis shows that the three mentioned steps – generation, aspect selection, weaving – can be reduced to the concept of model transformation.

The following steps are the same as for standard UML tools. The developer has to put further work in the low-level model since the high-level model is not detailed enough to generate a complete low-level model. Then, the UML tool generates source code and makefiles. Again, the developers can add new source code. Finally, the software system can be built, tested, and deployed. The deployment itself is modeled in the low-level design, too. Thus, the UML tool can be utilized to deploy the software on a set of machines.

Usually, a development process assigns duties to team members, i.e. a process defines *who* does *what*. The approach presented in this thesis assumes that the developers can be split at least into three categories. The first category has the lowest level of QoS proficiency. They can identify the QoS categories that are applicable to their system, but they do not know how to implement these QoS categories. For them the QoS-agnostic model and the QoS aspects in Figure 2.1 are not directly visible. Their view on the development process is depicted in Figure 2.2. They just start their model transformation tool. It translates the high-level design into a low-level design. The internals of this model transformation are not visible to them. A second-level-of-proficiency developer knows about the shaded QoS aspect box. This box contains all information required to weave QoS support into the implementation of a component. Hence, these developers are QoS experts and they know how to express the QoS-related implementation pieces as a reusable aspect. Third-level developers are able to dig into the transformation process that generates the low-level design from the high-level one. The sole purpose of these proficiency levels is to enable developers who are not very experienced in QoS to develop QoS-aware applications. Only second-level developers need to understand the details of QoS realization.

2.2.1 Model Driven Architecture

The proposed process fits with the Model Driven Architecture (MDA) [OMG03a] initiative of the OMG [OMG03b]. The MDA is an approach to separate business or application logic from underlying platform technology. Therefore, the MDA proposes two models:

- a platform independent model (PIM), and

- a platform specific model (PSM).

The PIM makes no assumptions about platform aspects such as middleware, GUI framework, operating system, programming language etc. Instead, the PIM models the application in terms of the application domain. A model transformer maps the PIM to a PSM, which is a model of the implementation. This means, the PSM depends on the selection of programming language, middleware, etc. Hence, the high-level model shown in Figure 2.1 can be treated as a PIM and the low-level model can be treated as a PSM.

However, the reasons for proposing these two kinds of models are different. The original goal of the MDA is to prevent intellectual property from being tied with a specific platform. Via the model transformation it is (in theory) possible to map an existing application (i.e. its PIM) with little effort on a new target platform. In contrast, the goal of this thesis is to tackle the complexity of QoS realization via the model transformation step. The PIM specifies *which* QoS properties apply to which component. The PSM shows *how* these QoS properties are realized on a specific target platform. Therefore, it is the task of the model transformer to map the QoS specification in the PIM to the corresponding QoS realization in the PSM. Of course it is very advantageous that one modeling language can be used to create PIMs of applications independent of the selected QoS-aware middleware platforms. However, the MDA goal of migrating existing applications easily from one platform to another is not envisaged by this thesis.

2.3 Structure of the Thesis

This thesis consists of four parts. The first part has covered the introduction (see chapter 1). It elaborates on the motivation for the research and provides a brief overview of focus and contribution (see chapter 2).

Part II discusses model transformation in detail. A general introduction to model transformation is given (see chapter 3), followed by foundations and related work (see chapter 4). Chapter 5 discusses the application of graph transformation to the MDA. Chapter 6 shows how round-trip engineering can be achieved in model-driven development with model transformation based on graph transformation theory. Finally, chapter 7 presents a visual rule-based transformation language, that is built on the theory developed in the previous two chapters, and chapter 8 elaborates on the tool chain developed throughout this thesis with a special focus on model transformation. In summary, part two presents the technology und tools on which the development process is built.

Part III presents a modeling language for the platform independent modeling of QoS-enabled distributed applications and discusses how the modeling language is to be used. The first chapter of part III provides an introduction to QoS and modeling of QoS-enabled applications (see chapter 9), followed by a chapter on related work and foundations (see chapter 10). Chapter 11 presents PIQML, a modeling language for QoS-enabled distributed applications. Chapter 12 shows how to build UML-based PSMs for different target platforms. Finally, chapter 13 shows how the transformation techniques developed in part II are to be used to transform QoS contracts from their PIM specification to their PSM realization.

Part IV of this thesis is dedicated to outlook and conclusions. Chapter 14 summarizes the findings presented in this thesis. Finally, chapter 15 discusses potential future work.

Part II

Model Transformation

Chapter 3

Introduction

In model-driven development models are the main assets of software development. Therefore, development tools have to act on models instead of source code. These tools can be divided in model editing tools and model transformation tools. Model transformers are the replacement for source-code-centric compilers and refactoring tools. Transformers read a model that adheres to some modeling language and write a model that may adhere to another modeling language. In special cases the modeling language of the input and the modeling language of the output are equal. This is, for example, the case for model refactoring tools [FBB+99, SPTJ01]. In the context of the MDA the input language is different from the output language. Furthermore, the input language resides on a higher level of abstraction than the output language. MDA model transformers can be compared with compilers, which transform a high level programming language into a machine executable language. While MDA model transformers work on models, compilers act on source code and binary code.

The following chapters discuss the techniques required for the implementation of model transformers. The presented concepts apply to MDA-style transformers, i.e. input and output languages differ. However, with little modifications the results can be applied to model refactoring, too. A short discussion of this idea can be found in chapter 15.

The purpose of the model transformation in the context of this thesis is twofold. First of all, the high-level design (PIM) has to be transformed into a low-level design (PSM). This fact is common to all MDA-based development processes. Second, within this thesis the transformation step is used to incorporate QoS-specific design pieces and source code fragments (QoS aspects) in the PSM.

Figure 3.1 illustrates a development cycle. The development starts with a PIM. Then it is transformed in three steps into a PSM. It is a definitive goal of this thesis to divide the transformation in these three steps since that isolates QoS-related artifacts. The first step, which creates a QoS-agnostic PSM, is aware of the fact that QoS exists. However, this transformation does not understand the semantics of the single QoS categories. Its task is to map the PIM components, their ports, interfaces, protocols, and connections to PSM concepts. The next two transformation steps are QoS-related. They are aware of the different QoS categories and their semantics. Step number two selects the QoS aspects that are to be used. Step number three weaves the output of step one with that of step two yielding the PSM.

Transforming models is a complex programming task. If a new or modified transformation is only needed when a new target platform is introduced then the complexity of this task is less crucial. However, in reality the transformation will be subject to more frequent changes and

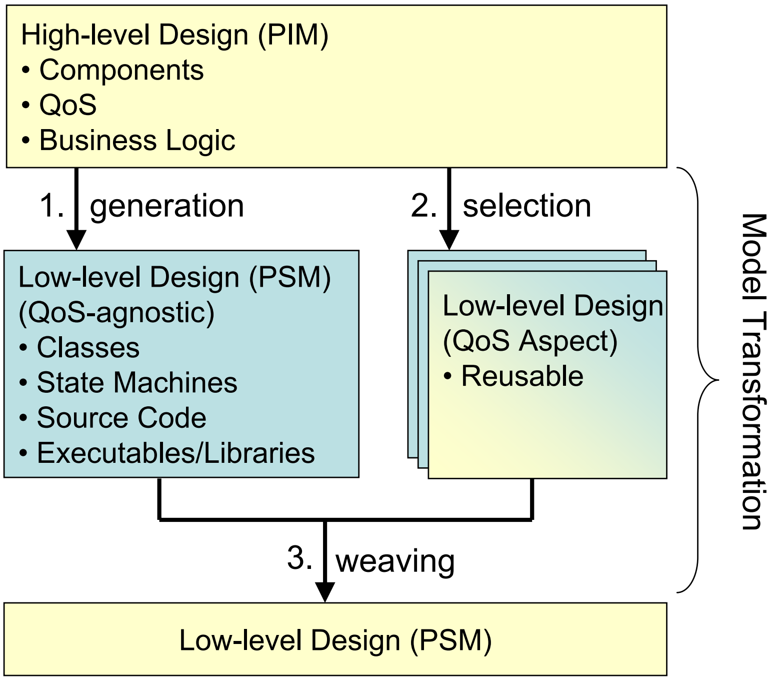


Figure 3.1: Model transformation overview

additions. For example, every development team has its own subset of design principles and patterns. The model transformation should take these into account when generating the PSM, hence, the platform specific design of the application.

3.1 Overview

Part II of this thesis is organized as follows. Chapter 3 provided an introduction and continues with requirement analysis. The following chapter explains foundations and related work. The chapter starts with a close look at the terms *model* and *modeling language* followed by a brief presentation of the de facto standard modeling language UML. Furthermore, important standards such as the Meta Object Facility (MOF) [OMG02d] and techniques for model interchange are presented. The next section discusses the theory of graph transformation since some transformation techniques depend on this theory. Then, existing approaches to model transformation are presented.

Chapter 5 shows how graph transformation can be used for MDA-style model transformers. The first section formalizes the UML. The following sections show the shortcomings of standard graph transformation theory when applied to the MDA and how it can be adapted to the MDA.

Chapter 6 explains how round-trip engineering in the context of the MDA can be performed if the model transformation is based on graph transformations. The chapter starts with a comparison of the terms *round-trip engineering* and *reverse engineering*. Then the concept of *forward-merging* is presented which supports round-trip engineering without reverse engineering. Finally, the role of diagrams in round-trip engineering is discussed.

Chapter 7 presents the visual rule-based model transformation language Kafka that has been developed throughout the work on this thesis. The first two sections describe the notation of the Kafka transformation language. The next section shows how Kafka is related with the graph transformation theory as developed in chapter 5 and that transformations developed with Kafka automatically support round-trip engineering. The following section explains how Python expressions can be embedded in Kafka transformations. Kafka is a rule-based language. Hence, the next section describes how these rules are orchestrated to form a complete model transformer. The chapter closes with an analysis of the algorithm that is used to execute a Kafka transformation.

3.2 Requirements

Every new QoS category requires changes to the model transformation. For example, a replicated service has a design different from that of a service with special security constraints. Furthermore, each QoS category can be tackled with different designs. For example, a service with high availability can either use replication with hot-standby or cold-standby. The chosen solution will for sure affect the design of the realization of a service as well as the choice of QoS mechanisms. Hence, it is quite likely that the model transformation is subject to frequent customizations. This yields the first requirement.

Requirement 3.1. Model transformations must be easily customizable.

Modeling languages such as the UML are built on a non-trivial meta model and a large set of well-formedness rules. The UML 1.4 meta model consist of over 110 meta classes and every meta class has a set of well-formedness rules. The next major version of UML [OMG03e, OMG03f] further increases the complexity of the meta model. In order to implement model transformations detailed knowledge of these meta model and well-formedness rules are required. However, most users of the UML are familiar with the graphical notation only. When using standard UML modeling tools, developers do not get in touch with the meta model and the well-formedness rules are automatically checked by the tool, too. Hence, a large gap separates an experienced UML user from a person capable of understanding the respective meta model. Ideally, the meta model could be shielded from the developers. In this case the transformer is constructed visually and stays within the notation of the modeling language.

Requirement 3.2. Model transformations must feature a notation that builds on the notation of the PIM modeling language and the PSM modeling language.

Transformations of QoS-aware application models can be decomposed in two parts. One part handles the mapping of PIM components, interfaces, connections, etc. to the respective PSM concept. The second part is QoS-aware. It transforms the QoS specification into code and design that deals with the realization of the QoS. Modularization in complex software projects is in general a good idea. If the implementation of both parts is mixed, the typical problems would arise:

- the responsibilities among developers are not clear,
- changes applied to satisfy one QoS category can harm the transformation of others,
- it is difficult to judge for which QoS category a certain transformation rule has been introduced,
- the more developers work on the transformer the more increases the probability of conflicts and the administrative overhead.

Requirement 3.3. The implementation of a transformer must be modularized. Modules should either be QoS-agnostic or handle a special QoS category.

Modern development processes are usually iterative. That means they pass the sequence of requirement analysis over design to implementation several times. With each iteration the product is enhanced with new features or restructured. Agile processes [Agi02] and especially Extreme Programming (XP) [Bec99, Fow01] feature very short iteration cycles. The MDA imposes a constraint on the development process: The sequence of PIM modeling, transformation, PSM modeling, and implementation has to be part of the overall process.

This imposes a problem on the transformation. The PIMs are usually not powerful enough to describe every detail of the application. Hence, the PSM is rather a design skeleton enriched with fragments of source code. The developers have to complete the design and source code where it could not be generated from the PIM. This results in a modification of the generated PSM. During the next iteration, developers may change the PIM and trigger the transformation again. This transformation would overwrite the PSM and all changes applied to it before would be lost. This is not acceptable.

Requirement 3.4. The transformation must support an iterative development process. Additions and changes applied to the PSM during an iteration should be reintegrated after a subsequent transformation.

Chapter 4

Foundations and Related Work

This chapter presents the theoretical foundations and industry standards on which the new model transformation technique depends. Furthermore, other approaches to model transformation and their pros and cons are discussed. The chapter starts with an explanation of the terms model, modeling language, and associated industry standards. The following section elaborates on the mathematical background for graph transformation, since the presented transformation technique depends on it. The chapter closes with an overview of related work.

4.1 Models

The term *model* stems from natural sciences, especially theoretical physics. The goal of a model is to provide a mathematical, hence precise, description of a rather complex system. The model does not necessarily capture the entire behavior of this system. However, it should at least allow the prediction of some behavioral or structural aspects. For example, the atom model of Niels Bohr [Boh13] is a model of an atom. The model covers structural and behavioral aspects of an atom. The model says that an atom consists of a core and a set of electrons and the distance between the electrons and the core is fixed. This is a structural description. The behavioral description says that the electrons circle around the core. The structural aspect of the model is well suited to explain the different energy levels of the electrons that can be observed in experiments. At the same time the model is wrong, since electrons do not circle around the core in orbits as proposed by the model [Tip02]. Hence, the model allows predicting certain aspects of nuclear physics but certainly not all.

A model is complete, if every experiment in the modeled system can be precisely predicted via a computation based on the model. In computer science, a complete model is called *executable*. This means, the model contains enough information to execute it on a virtual or physical machine. However, executable models are still subject to intensive research and it has yet to be proven that executable models will be significantly easier to handle than source code.

In physics, the modeled system is obviously nature, i.e. atoms, electrons, quarks, etc. In computer science there are at least two different systems a model can describe. First, a model can model the implementation. If the target language is object-oriented, such a model describes objects and their interaction. Second, a model can describe the application in terms of its application domain. In this case the model describes, for example, customers, bank accounts, products, orders, documents, and business rules. Ideally, one model covers both worlds. Object-oriented systems seemed to fulfill this promise. Bank accounts and customers can be modeled as objects. Business rules can be represented as constraints. These concepts are then mapped one-to-one onto an object-oriented programming language. In this case a model can at the same time model the application in its application domain and the implementation.

This idea worked out well for some application domains, for example graphical user interfaces. Windows, buttons, and text editors can be easily treated as objects. Hence, a one-to-one mapping between application domain and implementation language exists. In other cases the idea is doomed to fail. Especially the realization of QoS properties causes problems. For example, introducing replication into an object-oriented implementation will add artifacts (i.e. classes and methods) for synchronization between the replicas. These artifacts have no correspondence in the application domain. Hence, a model of this implementation is not at the same time a model in terms of the application domain.

A consequence is that two different models exist: one in the application domain and one in the realization domain. Software engineers tried to create and investigate methods and techniques for bridging the gap between the different models. In the seamless approach developers must bridge the gap with a sequence of model refinements. Developers start with an application domain model. This model is modified step-by-step until it is an implementation model. The problem of this approach is that the semantics of such intermediate models are not always easy to understand because the intermediate models are built on concepts of both modeling languages. If the two domains (application and realization) are substantially different, a stepwise transition from one model to the other is impossible.

Therefore, the Model Driven Architecture (MDA) of the OMG proposes a different solution. In contrast to the seamless approach, the MDA proposes to gap the bridge between both modeling domains in one big step. Furthermore, this step - a model transformation - should be automated. In summary, the MDA proposes to transform a model of the application domain into a model of the realization domain.

4.1.1 Modeling Languages

Each model must adhere to a modeling language. Otherwise it would only be a meaningless drawing. A modeling language consists of:

- abstract syntax,
- well-formedness rules,
- notation, and
- semantics.

The *abstract syntax* is the model-equivalent to the structure of a syntax tree as yielded by parsers of programming languages [ASU86]. The abstract syntax defines a data structure that is able to hold every model adhering to this language. The *well-formedness rules* further constraint this data structure. For example, a rule may enforce that objects must not use private methods of other objects. Such rules exist for syntax trees of programming languages, too. The *notation* has no equivalent in traditional programming languages. Models can even have multiple notations, graphical ones and textual ones. A notation provides a view on the model. In contrast, traditional programming languages have no explicit notation. Their syntax implies their notation. This is the reason why the syntax of a model is called *abstract* syntax.

The abstract syntax does not imply a notation. The *semantics* of a modeling language link the concepts of the modeling language to a domain (application or realization domain). Usually, the semantics are provided in an informal way. The drawback is that this leaves room for interpretation and variations. Another option of expressing semantics is to reduce the concepts of the modeling language to a well understood and defined mathematical model. For example, many research groups [RKR$^+$00] devised techniques for mapping UML state machines on Petri Nets to specify precise semantics for state machine models.

4.1.2 The Unified Modeling Language

The Unified Modeling Language (UML) is currently the most popular object-oriented modeling language. The UML is an OMG [OMG03b] standard. At the writing of this thesis the most recent UML standard was UML 1.5 [OMG03i]. All UML 1.x versions are defined via one large meta model (see section 4.2), a set of well-formedness rules, and a textual description of semantics and notation. UML 1.x provides concepts for the modeling of structure (i.e. classes, components, packages, etc.) and behavior (i.e. state machines, interaction diagrams, sequence charts, etc.). A more detailed description of UML can be found in [Fow97].

Missing formal semantics are one of the major UML shortcomings. If the semantic of a model are not exactly defined, code generators and model transformers have to interpret the model in one way or the other. This can lead to unexpected results. In contrast, programming languages and their semantics are usually better defined or backed up by a reference implementation. Hence, switching from one compiler to another will usually not change the behavior of the generated machine code. Generating source code from one model using different UML tools will lead to quite different results. The next version of UML does not seem to fix this problem. Therefore, some researchers are working on a precise UML [The03]. Once, the UML has precise semantics, it might be possible to specify executable models in UML, i.e. machine code can be automatically generated by a tool chain from a UML model. Such models are called *executable models*.

Another interesting aspect of UML is its extension mechanism. The UML is extensible via *stereotypes*. A stereotype can be attached to every model element. It adds a special meaning to this model element. For example, an $\ll$ EJB $\gg$ stereotype can be attached to a UML component indicating that it is an EJB component. A set of stereotypes is grouped to a so-called *profile*. UML profiles are widely used to adapt the UML to various programming languages or frameworks. However, this extension mechanism is often abused. Some profiles try to *redefine* the meaning of UML model elements, but by definition profiles can only *refine* them. This observation led to the idea of splitting the UML meta model into an infra-structure and a super-structure. New modeling languages can be based on a UML infra-structure. This avoids a redefinition of existing UML elements as done by some UML profiles. At the same time, basic concepts of UML can be reused and do not have to be reinvented.

The new UML 2.0 standard will be split into an infra-structure [OMG03e] and a super-structure [OMG03f]. At the writing of this thesis, the final version of UML 2.0 has not been adopted as an official standard by the OMG. However, only minor changes are expected for the final version. Time will tell whether the idea of a family of modeling languages based on the UML infrastructure will become reality.

4.2 Meta Object Facility

All languages must somehow specify what a text or model has to fulfill to comply with the language. For textual languages a grammar defines a language. This grammar is in turn specified using another textual language, for example EBNF. The same concept applies to modeling languages.

A model can be represented as the instance of a *meta model*. The meta model is the model-counterpart to the grammar of textual languages. It defines the modeling language. The meta model is in turn a model, hence, it can be treated as the instance of a meta-meta model. This could be repeated until infinity. However, the concepts of some meta^n model will be the same as the concepts of the meta^{n-1} model because the minimal set of concepts has been reached. Even more likely, some meta^m model with $m < n$ will have only such generic concepts that this meta^m model does not provide any benefit. Hence, an infinite sequence of meta, meta-meta, and meta-meta-meta models wont make much sense.

The Meta Object Facility [OMG02d] of the OMG defines four models. The first one (M0) is the running system. It is an instance of the M1 model, which describes the design of the system. The M2 model is in fact a meta model. It describes the modeling language. The M3 model is a meta-meta model. It is a language for meta models just like the EBNF is for textual languages.

The following example illustrates the idea of one model being an instance of another model. The simple class diagram depicted in figure Figure 4.1 can be viewed as an instance of the UML meta model. This is shown in figure Figure 4.2. Each box is an instance of a class of the meta model, i.e. the instance of a meta class. Both figures represent the same thing using different notations. The first figure is much easier to understand, hence, it is targeted at the human reader. The second figure shows the data structure that represents the model as instance of its meta-model.

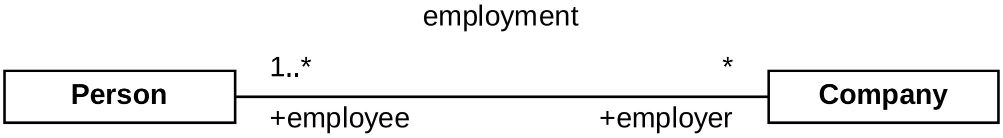


Figure 4.1: A sample class diagram [Bre02]

The MOF is especially important for model-related tools which are independent of a concrete modeling language. Via a M2 model such tools can read the specification of a modeling language as part of their input. This is comparable to parser generators. They read the specification of a textual programming language and yield a parser for this language. The problem of such meta-level tools is that they do not get any information about the semantics of the modeling languages. The M2 model just describes the structure of the modeling language and not its semantics.

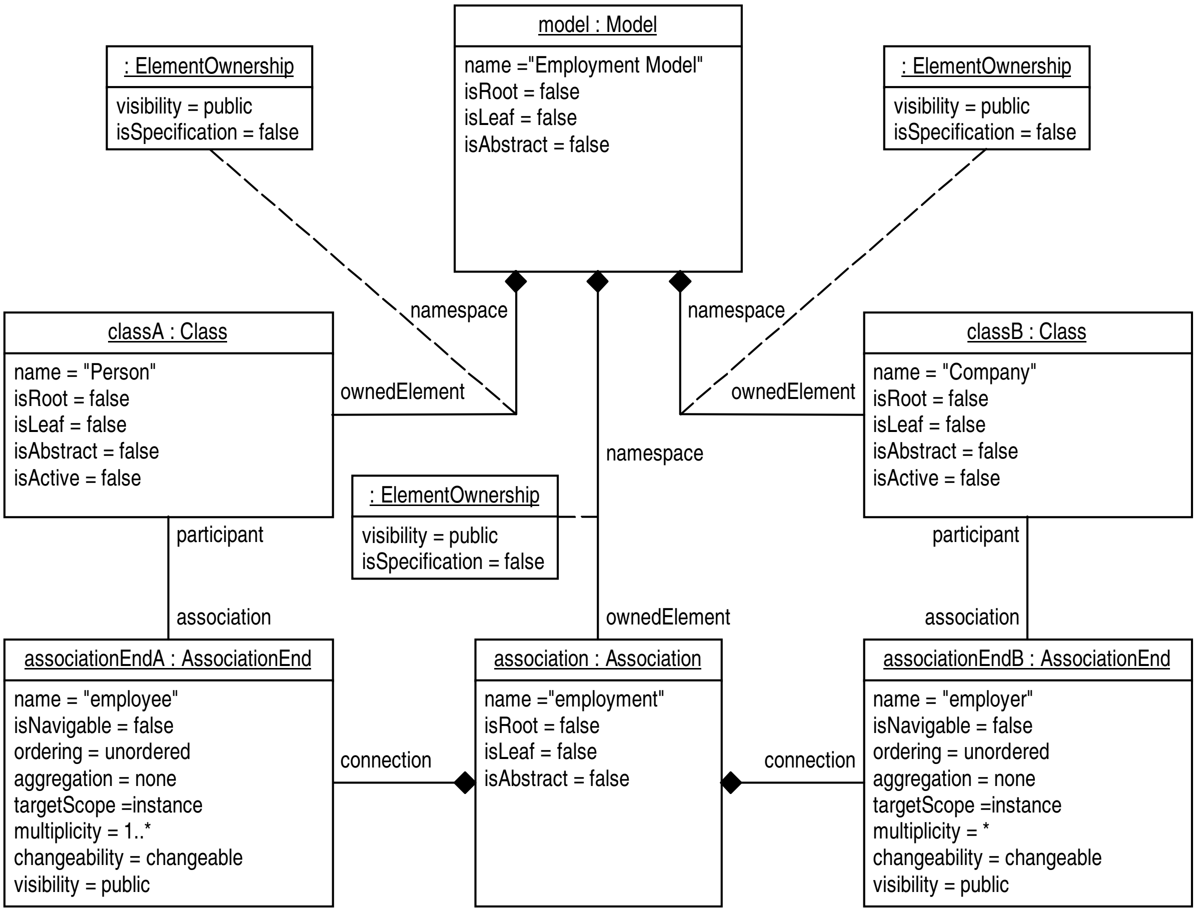


Figure 4.2: The same diagram depicted as an object diagram [Bre02]

4.2.1 MOF-compliant UML

Some artifacts of the UML meta model cannot be mapped directly to MOF. First of all, the UML meta model features association-classes. This is a hybrid of an association and a class and is not supported by the MOF. Furthermore, some associations in the UML meta model do not have role names attached to them. This is not allowed by MOF, too.

As explained in [Bre02] an association class can be transformed into one class and two associations. Another possibility is to shift the attributes of the association class to one of the associated classes. In both cases the result is a modified meta model with the same expressive power but without association classes. If a role name is missing, it can be easily substituted by the name of the associated class.

The UML specification itself provides a solution for these problems. The chapter titled “UML Model Interchange” in [OMG03i] describes a variation of the UML meta model. The expressive power of the meta model has not changed. Just the two problems - association classes and missing role names - have been addressed as described above. The rational for this interchange model was to align the UML meta model with the MOF standard.

4.3 Model Interchange

Serializing models is a frequent problem. If the meta model is MOF-based, then an XML DTD can be automatically derived from the meta model using the XMI [OMG03j] standard. XMI is usually used to exchange models between the modeling tool (i.e. the editor) and the model transformation tools. This allows for a loose coupling of the modeling tools and the model transformation tools. A drawback of XMI is that it does not contain diagramming information. Hence, after export/import all diagramming information is lost. The developer has to rearrange the elements on the screen which is a time consuming and cumbersome job. Nevertheless, many approaches to model transformation are based on XSLT [W3C99b] acting on XMI documents, for example: [DHO01, PBG01, PZB00, VVP02, KH02]. An alternative to XMI import/export is to perform the transformation process directly in the modeling tool. In this case the transformation works directly on the data structures of the modeling tool. Hence, the modeling tool can adapt the diagrams as the model is transformed. This option is discussed in more detail in chapter 8.

4.3.1 UML as a Tree Structure

XMI is based on XML and every XML document has a tree structure. Hence, to serialize a model with XMI a spanning tree has to be calculated. This can be achieved by pruning all edges that are not instances of a meta composition (depicted with a black diamond in the meta model). Cyclic compositions are not allowed by the MOF. Furthermore, a model element is part of at most one composite model element. Therefore, the pruning yields a set of trees G_a for a model $G = (V, E)$ where V is the set of vertices (i.e. instances of UML meta classes) and E is the set of edges (i.e. instances of UML meta associations).

$$\begin{aligned} G_a &= (V, E_a) \\ E_a &= \{\ e \mid e = (v_1, v_2) \in E, e \text{ is a composition } \} \end{aligned}$$

A spanning tree can be constructed by introducing a root element.

Definition 4.1. $G_{span} = (V_{span}, E_{span})$ is a *spanning tree* for $G = (V, E)$.

$V_{span} = V \cup \{root\}$

$E_a = \{\ e \mid e \in E, e \text{ is a composition } \}$

$E_{span} = E_a \cup \{\ (root, v) \mid v \neq root, \neg\exists\ v_2 : (v_2, v) \in E_a\ \}$

The importance of G_{span} for model transformations is twofold. Some programming languages are especially well suited for the handling of tree based data structures. Two of the model transformation approaches discussed in section 4.5 rely on the tree structure. For example

XSLT based transformers such as [PBG01] use the tree structure for pattern matching. Others such as UMLAUT [Tri01, HJGP99, HPP00] use the tree to iterate over the model.

Another advantage of the spanning tree is that it can be used for automatic garbage collection. For example, if a model transformation script deletes a class from a UML model, all remaining model elements belonging to the class can be deleted automatically, too. Otherwise the script would have to delete every attribute, operation, parameters etc. explicitly.

4.4 Graph Transformation

As detailed in section 5.1 a UML model can be treated as a directed graph with labeled nodes and labeled edges. Hence, transforming UML models is a special application of graph transformations. Graph transformations are a well investigated problem with a sound theoretical background [AEH+99]. Graphs are transformed by applying *graph transformation rules*.

Definition 4.2. A *graph transformation rule* is defined as $r = (L, R, K, map_{LR}, map_{KR}, appl)$. L and R are two graphs, called left-hand side and right-hand side. K is a sub-graph of L and the projection map_{KR} maps all vertices and edges of K on R. Therefore, K has an occurrence in R, too. The projection map_{LR} maps some vertices of L on R. Finally, $appl$ is an application constraint. If the constraint is satisfied then the transformation rule can be applied, otherwise not.

A transformation engine applies transformation rules to graphs. The result of the transformation is called a derivation of the initial graph.

Definition 4.3. The application of a transformation rule r on a graph G is notated as $G \Rightarrow_r G_2$. The graph G_2 is a *direct derivation* of G with respect to r.

An application of a transformation rule can be split in several steps. The first task is to find an occurrence of L in G. The following steps have the goal of replacing L in G with R. Hence, this process is called graph rewriting. The left-hand graph L is searched and replaced (rewritten) with R.

All vertices and edges of L except those in K are deleted. In other words: L is pruned up to K. The next step glues R with G. The graph K has an occurrence in L, hence G, and in R. The gluing process maps both occurrences. Hence, K determines where the graphs G and R are glued together. The pruning of L may have left over some dangling edges. If an edge used to be connected with a vertex v_L in L and the projection map_{LR} maps it to a vertex v_R in R then the edge is reconnected with v_R. Otherwise the dangling edge is removed from the graph.

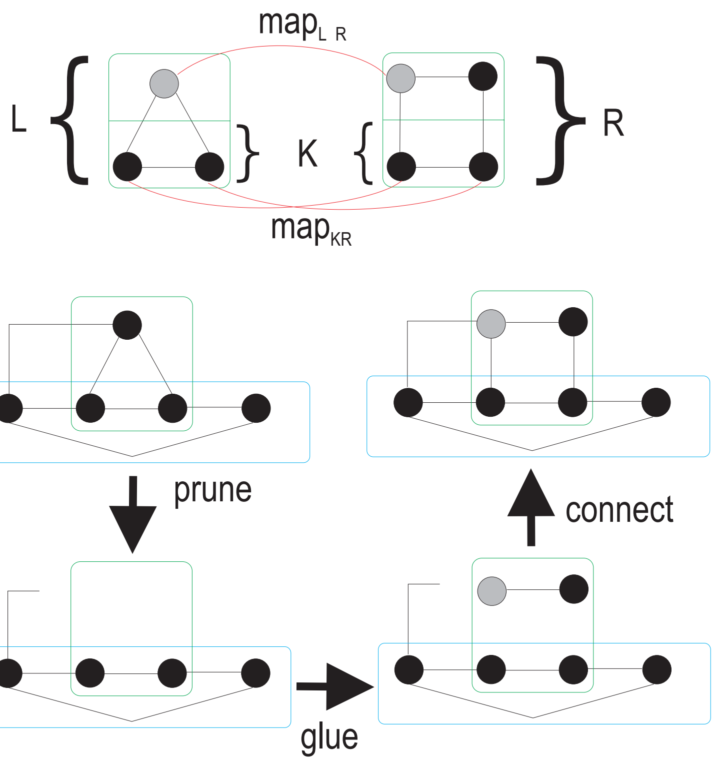


Figure 4.3: Application of a transformation rule

The application of a transformation rule is illustrated by Figure 4.3. The first two graphs represent the transformation rule expressed as graphs L, R, K, and the two mappings map_{LR} and map_{KR}. The other three graphs show the transformation process step by step. First, L is removed up to K (prune). Then R is glued with the pruned graph. Finally, the missing or dangling edges are reconnected. A more exhaustive discussion can be found in [AEH^{+}99]. The following observation can be deduced from the described transformation algorithm.

Observation 4.1. The structure of sub-graph K is preserved by the application of a transformation rule.

The observation follows from the fact that K is a sub-graph of L and has an occurrence in R. If the transformed graph is labeled, the transformation must not necessarily preserve the labels of K. Hence, just the structure is preserved.

4.5 Transformation Languages

Several general purpose programming languages and specialized transformation languages have been applied to the domain of model transformation. The different approaches can be classified in different ways. For example, [CH03] proposed a feature-based classification. In the following sections, the approaches are classified by the kind of programming paradigm used, i.e. functional, imperative, rule-based etc.

4.5.1 Functional Languages

Literature features several approaches to model transformation languages. The idea presented in [HJGP99] proposes the use of a functional programming language. Following this approach, developers have to develop a set of transformation operators. These operators can be concatenated using the $filter$, *reduce* and *map* functions. The following example taken from [HPP00] illustrates this:

$$(\mathit{map}\ \text{removeAssociation}) \circ (filter\ \text{isClient})\ \text{allElements}$$

This expression iterates over all elements of the model. The $filter$ function uses an operator to select the model elements that are to be translated. Finally, the *map* function passes all selected elements through a transformation operator.

One advantage of this approach is that the developer does not have to care about how to traverse the model. As shown in subsection 4.5.3 this is exactly the drawback of object-oriented languages. Using a functional language, complex transformations can be easily constructed reusing existing operators. Another advantage of operator composition is that developers using an operator do not need to understand how the operator is implemented. Hence, developers can be divided in two groups. The first group simply reuses or parameterizes existing operators. The second group is concerned with the development of operators.

However, the entire approach has two weaknesses. The side effects of some operators cause a problem, since functional programming ideally avoids side effects. The same problems appear in other applications of functional languages, too, for example when dealing with efficient IO. For this purpose Haskell [PC03] (a functional language) uses a concept called monads [Jon01] to cope with the problem. Another problem related to the first one is the save iteration over all elements. The transformation engine iterates over the spanning tree G_{span} (see definition 4.1) to iterate over a model. The model – including G_{span} – is modified during iteration. Therefore, it is difficult to ensure that all elements are visited exactly once.

4.5.2 Pattern-based Languages

The standard serialization format for models is based on XML. Consequently, some researchers [DHO01, PBG01, PZB00, VVP02, KH02] suggest transforming the XML document instead of the graph structure. The rational of their proposal is that a transformation language for XML already exists: XSLT. XSLT [W3C99b] is a W3C standard. The original purpose of XSLT was to transform arbitrary XML documents into XSL documents. Such an XSL document is just another XML compliant document. XSL documents put the emphasis on the layout. They can be compared to PDF or Postscript. Usually, XML documents emphasize on the content and are ideally free of layout information. Using an XSLT script the content of the XML document can be enriched with layout information and displayed in a browser. However, XSLT can be used as a general purpose transformation language for XML documents.

XSLT uses the concept of patterns. It traverses the XML tree structure and tries to match its pattern at every node of the tree. If the pattern matches, a certain transformation rule is applied. While this concept works very well for trees, it is difficult to adapt it to arbitrarily shaped graphs. Furthermore, the problem of sub-graph matching in arbitrarily shaped graphs is known to be NP-complete [GJ90].

XSLT can operate in two modes. One mode is used to generate another XML document. In the other modus the XSLT produces a plain ASCII or UTF8/16 text file. When utilizing XSLT for model transformations, the first modus is required. The XSLT processor reads the XMI document (i.e. a model) and produces another XMI file. In [Sar02] the second modus has been used for code generation. Here, the XSLT processor generates source code files from the XMI.

The idea is appealing on first sight because it is based on standard technology, but it has severe drawbacks. XML documents are by definition tree structures only. It is possible to cross link elements outside the tree structure using new W3C standards such as XPointer [W3C02] or XLink [W3C01]. However, XMI does not use them. Using the XPath [W3C99a] standard, it is possible to traverse edges of E that are not in E_{span}. The usage of XPath is quite complicated. This implies that it is very difficult for XSLT scripts to take those edges of G into account that are not part of E_{span}.

This problem is very evident when transforming state machines, activity diagrams, or sequence diagrams. When serializing such diagrams (respectively their model) using XMI, the most important information is not represented in G_{span}. The following example uses state machines to illustrate the problem. The edges E_{span} of a state machines model will only tell which state is owned by which composite state. E_{span} cannot reveal which states are connected via state transitions. The meta associations used to model the incoming and outgoing state transitions are not compositions. For this reason the instances of these meta associations are not part of E_{span}. This limits XSLT-based model transformation to a subset of UML. XSLT scripts can be useful to transform class diagrams, component diagrams, or deployment diagrams. Their most important information is present in G_{span}. Transforming behavioral diagrams is rather difficult.

[PBG01] provides another critical discussion of XSLT in the context of model transformation. The authors argue that XSLT is powerful enough for general purpose transformations but too difficult to use. To circumvent this problem, they propose a front end language that is automatically translated to XSLT. In this approach, XSLT is only the execution engine for transformations.

4.5.3 Object-oriented Languages

Currently, object-oriented languages are the best choice when general purpose programming languages are needed. Applying them to model transformation is straight forward. The modeling tool must provide an object-oriented API that grants access to the model. The drawback of OO languages is that the programmer has to write code for traversing the data structures.

Traversing trees or even arbitrarily shaped graphs is a non-trivial task. The result is usually deeply nested loops and the increased possibility of bugs such as infinite loops or infinite recursions. Other approaches like functional languages (see subsection 4.5.1) or pattern-based languages (see subsection 4.5.2) hide the model traversal from the developer.

4.5.4 Hybrid Languages

Some programming languages tend to merge benefits of functional programming and object orientation. Two representatives of this approach are Python and OCL. OCL is an acronym for Object Constraint Language [OMG03i, OMG03d]. The OCL is part of the UML 1.5 specification. It has been developed to describe constraints on instances of models. OCL can be used on the meta level, too, because meta models are once again just models. In this case, the constraints are applied to the model itself instead of model instances.

OCL is a hybrid language merging functional and OO concepts. OCL features a syntax that is very close to that of object-oriented languages. Furthermore, the handling of data types such as sets, bags, strings, etc., is very close to that of OO languages. Other constructs such as $forall$ and *select* are conceptually close to the *map*, $filter$, and *reduce* functions.

OCL has been designed to work on models. It does not come by surprise that it seems most suitable for the problem of model transformation. The hitch is that OCL is a constraint language and designed to be free of side effects. However, model transformation inevitably causes side effects. The UMLAUT transformation framework [Tri01] is a research project that builds model transformations on top of the upcoming OCL2 [OMG03d] and the action semantics standard [SPH$^+$01, OMG03d]. The action semantics represent an extension of OCL2. They allow OCL2 scripts to add and remove objects and to change their properties. Action semantics have been designed to formally specify the behavior of a system on the M0 level. Just like OCL itself, they can be applied to the meta level, too.

The VisualOCL project [BKPPT01, KTW02] equips OCL with a graphical notation. This may ease the creation and understanding of OCL because its textual notation is very compact. Nevertheless, VisualOCL's notation does not provide new concepts. It is just another notation of OCL. In [BKPPT02] VisualOCL is combined with graph transformation theory. This way OCL is extended with operational semantics. Thus, it can be used for model transformations.

Python [Pyt03] is an object oriented language that has been extended with elements of functional programming. For example, Python features the *map*, *reduce* operators. *Lambda* functions can be used, too. Furthermore, Python supports Haskell like "list comprehension". These features are very useful when implementing model transformations. Very often a transformation deals with a list of model elements. This list must be mapped or reduced. Python's functional language elements allow doing this with very few lines of code. For example the code fragment

```
lst = []
for x in uml.getAll():
    if x.name() == "FindMe":
        lst = lst + [x]
```

can be reduced to

```
lst = [ x for x in uml.getAll() if x.name() == "FindMe"]
```

Python is used as a scripting language in the modeling tool developed throughout this thesis (see chapter 8 for details). Python is used in commercial tools, too. For example, the MDA tools of Interactive Objects [Int03] build on Python when it comes to scripting.

4.5.5 Graphical Languages

A drawback of textual languages for graph transformations is that text is usually not the ideal media for graphs. Instead, graphs inherently have a non-textual, hence graphical, notation. Some approaches have been devised to exploit the advantage of a graphical notation.

AGG [TFS03, EEKR99, Tae99] is a tool for editing and transforming graphs. A graph editor supports the editing of graphs and mappings between graphs. Since a graph transformation rule is composed of three graphs and two graph mappings, it is possible to model transformation rules in AGG. Applying AGG to model transformations is in theory possible although AGG has not been designed with UML in mind. AGG does not make use of the fact that UML already has a graphical notation. Instead, every model would be represented as a graph where instances of meta classes are depicted as nodes. Instances of meta associations are shown as edges between the nodes. Since AGG supports higher-order graphs it would even be possible to handle association classes in the meta model. Higher-order graphs allow for edges between edges. This way, nodes and edges can be handled in a uniform way [LB93]. If a graph transformation system cannot handle higher-order graphs, workarounds are possible (see subsection 4.2.1).

Some researches enhanced the UML itself with graph transformation capabilities. Story diagrams [FNTZ98] are one example. A Story Diagram is a special kind of action diagram. It shows a set of objects (M0 level) and their links and it shows how the objects change over time. Story Diagrams show how links between objects are created or removed, how objects are created or destroyed or how the attribute values of the objects change. Hence, Story diagrams are a way of modeling the behavior of a system. The intention of Story Diagrams is not to work on the model level (M1). Instead, they apply the transformation steps to the object level (M0). This means that the transformation acts on instances on classes, not instances of meta classes. Therefore, transformations specify how a system behaves at runtime. Conceptually, this is comparable to the approach taken by AGG. Only the notation is different. Nodes are represented as UML objects and edges are shown as UML links between the objects.

The approach taken by [HE00] is a merge of UML object diagrams and graph transformations. In so far it is comparable to Story Diagrams. Just like [FNTZ98] they use ideas from graph transformation theory to model the behavior of a system. Consequently, the approach acts on the M0 level, too, just like Story Diagrams.

However, the MDA is not concerned with the M0 level. Transformations address the models themselves, hence the M1 level. Conceptually, it is possible to *shift the meta levels* since a model can be treated as an instance of the meta model just like the M0 objects are instances of the M1 classes. Story Diagrams, the approach of [HE00], or AGG could be used for model transformation, too. However, a shift of the meta models would ideally be accompanied with a shift of the notation. An M1 instance of an M2 meta class has a special notation in the UML that depends on the meta class. For example an enumeration is shown as a box while an actor is depicted as a stickman. An M0 instance of an M1 class in contrast is always depicted as a rectangle disregarding of the M1 class. A shift of notation cannot be easily accomplished since the UML notation extension introduced for Story Diagrams works for UML object diagrams only. Therefore, the notational support of Story Diagrams is as weak as that of AGG. Whether a node is displayed as a rectangle (Story Diagrams, [HE00]) or circle (AGG) does not really matter. Figure 4.1 and Figure 4.2 illustrate the shift of notation. Both models show the same content. However, the second one uses the notation for M0 level objects while the first one uses the notation for M1 level objects.

All transformation techniques presented in this section *modify* graphs. This means a start graph is transformed several times to reach a final derivation. Usually, this final derivation still shares a common subset with the start graph. This is the case for all examples found in [AEH$^+$99, FNTZ98, HE00]. In [Cla01] this is called “in-place” editing. In the MDA it is less common to morph a PIM step by step until it is a valid PSM. Even worse, this is usually not possible at all since the PIM is based on another meta model. Recalling definition 4.2, the problem is the sub-graph K and its occurrence map_{KR} in R. L is a part of a PIM that should be translated into a corresponding PSM construct. R is a sub-graph of the PSM. K should have an occurrence in L and R but L and R are built on two distinct meta models. Therefore, K cannot have an occurrence in L *and* R at the same time.

VIATRA [VVP02, CHM$^+$02, VGP01] is another model transformation approach based on graph transformation. In contrast to the other approaches discussed in this chapter, VIATRA has been designed to transform a model from one modeling language to another one, i.e. it does not do in-place editing. The overall goal of VIATRA is to check the correctness and reliability of a design. Therefore, the design (expressed as a UML model) is transformed into a more precise model better suited for formal reasoning. For example, VIATRA has been used to transform UML state machines to Petri Nets. Once the transformation is done, existing tools for Petri Nets can be applied. The result of these tools is back-annotated to the UML state machine model. To achieve this back-annotation, VIATRA builds a reference graph between the input and the output model during the transformation process. VIATRA rules can be edited with a UML tool and a special UML profile. However, the notation of VIATRA rules uses the meta level shifting as described above. The left-hand side and the right-hand side of a VIATRA rule are displayed as UML object diagrams. Thus, the rule developer is directly confronted with the meta model of the input and the output modeling language.

4.6 Model Driven Architecture

In contrast to the model-related technologies discussed in this chapter, the Model Driven Architecture (MDA) [OMG03a] is not a concrete technology. It is more kind of a *vision* of the Object Management Group (OMG) [OMG03b] aiming at *platform independent* software development. The OMG has followed the idea of platform independent object systems for a couple of years. With CORBA [OMG02a] the OMG defined a middleware that allows objects to communicate disregarding of the platform they are running on and disregarding of their implementation language. However, only the interfaces of the objects are subject to a platform independent description in CORBA. The concrete implementation depends on the selected programming language and operating system. Very often, the implementation even depends on a concrete CORBA ORB (Object Request Broker). Hence, CORBA allows specifying interfaces in a platform independent manner but the behavior of objects is still *platform specific.*

Platform specific software artifacts are bound to concrete technologies which are subject to frequent changes. Operating systems for PCs and UNIX servers are updated every 3-5 years. Frameworks for distribution or graphical user interfaces even evolve faster. Software is a major investment for companies. If technologies changes very frequently, these investments will soon rely on outdated technology or they need continuous development to keep up with new technologies. The core of the MDA vision is to keep software (i.e. an investment) platform independent. An automated tool chain will then map the platform independent software description on a concrete target platform. If the target platform changes, only a new mapping is required and existing software will still work.

This vision needs more than just platform independent interface definitions as provided by CORBA. The behavior and interaction of objects must be captured in a platform independent manner. The MDA proposes the use of models to describe objects, their interface, behavior, and interaction. Modeling languages have been used for several years to specify and document large object systems. Different object-oriented modeling languages have been devised. The most popular ones have been unified and gave birth to the Unified Modeling Language (UML). The UML is a major building block of the MDA. Many MDA tools use UML or a UML derivation as their modeling language. In fact, modeling tool vendors consider MDA as an add-on to their UML tools. The other building block of the MDA is the Meta Object Facility (MOF). The MOF is important, because the MDA is not tied to a concrete modeling language. Every modeling language with a MOF-compliant meta-model can be used if it is not tied to a specific platform, i.e. programming language, framework, or operating system. Developers are supposed to use such a platform independent modeling language to create a *platform independent model* (PIM) of an object system.

This PIM is the most important software artifact. However, computers execute machine code and not PIMs. Thus, the platform independent model must be fed into a specialized tool chain. A model transformation tool transforms the PIM into a *platform specific* model (PSM). The PSM takes specialties of the target platform into consideration. For example, a PIM represents the software for controlling an industry robot. This PIM can be mapped to different target platforms, for example a small embedded system or a full fledged PC. On the embedded system, the control software can do busy waiting, because it is the only process executing on the system. On a PC, the control system should be implemented as an event-driven system since busy waiting would waste CPU power that can be used for other processes. This example illustrates how one PIM can be mapped to several PSMs. PIM and PSM can be subject to different modeling languages. Usually, the platform specific language has some special constructs dedicated to the target programming language (for example, C++ templates or C# delegates).

Another major difference between a platform independent modeling language and the platform specific modeling language is the *domain* from which they lend their concepts. The platform independent language usually uses concepts of the application domain. For example, a modeling language for enterprise systems may use concepts such as products, customers, documents, signatures, throughput, etc. In contrast, a platform specific language uses concepts of the realization domain, i.e. classes, operations, state machines. As a rule of thumb, the PIM describes *what* the software does, while the PSM describes *how* this is realized on a concrete platform. The way from a PSM via source code to machine code is well known from UML tools. Hence, the special part of a MDA tool chain is the model transformation that maps the PIM on its PSM counterpart.

The model transformation is among the most complicated techniques required for a MDA tool chain. The previous section already discussed the different approaches to model transformation. Implementing a transformation is already a very complex task. However, the best model transformation approach will not help if the transformation is inherently impossible. For example, one could have the idea of building a meta model for the programming language Java and to use it as a general purpose platform independent modeling language. Building such a meta model is technically possible. It can be derived from the EBNF grammar of Java. Implementing a transformer that maps, for example, such a Java-PIM on a C#-specific PSM is hardly impossible with the current state of the art. Microsoft has been working on a Java-to-C# transformer. Although both languages are quite similar, the result of the transformation is not complete and the quality of the generated code is questionable. There are many reasons why such a transformer is doomed to fail. First of all, current object-oriented programs are not free of side effects. A transformer would have to understand all possible side effects to generate C# source code that behaves exactly like the Java-PIM. Furthermore, a Java-PIM would already contain Java-specific optimizations or design decisions. For example, thread synchronization in Java and .NET are quite different. If the PIM has already been defined with Java in mind, it is hard to generate a solution optimized for .NET. The strange idea of a Java-PIM shows that it is very hard to come up with a platform independent modeling language that is detailed enough to model complex applications and still abstract enough to allow reasonable transformations.

The MDA proposes a shift of paradigms in software development. Developers should use models to express *what* the software system does and which constraints (i.e. QoS) apply. Tools care about *how* to realize such systems. The major challenges are to develop sophisticated platform independent modeling languages and associated model transformers.

Chapter 5

Graph Transformation

This chapter shows how graph transformation theory can be adapted to the special needs of MDA transformations. The goal is build the theoretical foundations for an easy to use rule-based visual transformation language. This language and its notation is discussed in chapter 7. Furthermore, the use of graph theory will enable round-trip engineering. Chapter 6 will exploit this feature in detail.

The reason for choosing an approach based on graph transformation rules, originates in practical experience. In the context of this thesis industry and research partners of an IST project used Python (an object oriented language) to implement model transformations. It turned out that implementing transformations this way was time consuming and error-prone even though the developers knew their meta model by heart. One major reason for mistakes was the traversal of models. Chances were high that either some elements were left out or an endless loop occurred. This chuckhole disappears when using rule-based languages since the engine applying the rules performs all the model traversals. This observation already led [HJGP99, HPP00] to propose functional languages for model transformation. The second source of errors was misinterpretation of the meta model. Using a visual notation that hides the complex meta model behind a more user-friendly notation solves this problem. Thus, a rule-based transformation language with a visual notation seems to be the optimal fit.

A careful analysis of a set of Python-based transformers revealed that they performed basically four kinds of actions:

- Search for a certain *pattern* in the PIM.
- Determine relevant elements of the current "PSM under construction", i.e. a pattern search in the PSM.
- Add, modify, or remove PSM elements, i.e. a certain pattern is applied.
- Bookkeeping of the already generated PSM elements.

Except for the bookkeeping, these steps are performed by applying graph transformation rules. A rule has a left-hand search pattern L. All matches of L in the start graph are computed. Then L is replaced by a graph R while an interface graph $K \subseteq L$ is kept invariant. Hence, results of graph transformation theory can be applied to the problem of model transformation.

5.1 Formalization

Applying graph transformation theory to the problem of model transformation requires an appropriate formalization of the terms model and meta model. Specifications of modeling languages usually contain a formal or semi formal definition of the language. Modeling languages can be defined by another model – the *meta model*. This led to the OMG's Meta Object Facility (MOF) (see section 4.2). It describes a *meta meta model* that can be used to model *meta models*. Thus, the MOF provides formalization on the meta-meta level.

However, the MOF is not an appropriate formalization for the algorithms discussed in this chapter. The algorithms build on graph transformation theory. Therefore, models – and meta models – have to be treated as graph structures with vertices V and edges E. The following definitions provide a formalization of these graphs.

Definition 5.1. A *meta model* $\mathcal{M}$ is defined as a directed labeled multi-graph $\mathcal{M} = (V, E)$ with $E \subseteq V \times V$.

V is the set of meta classes and E represents the meta associations.

The labels on V represent the meta class name. Names are just strings. Hence, they can be treated as integers. The labeling function is defined as $\ell_V : V \mapsto \mathbb{N}$ and every $v \in V$ has a unique label.

$\forall v_1, v_2 \in V : \; \ell_V(v_1) = \ell_V(v_2) \;\Leftrightarrow\; v_1 = v_2$

The labeling function for E is defined as $\ell_E : E \mapsto (\mathbb{N} \times K) \times (\mathbb{N} \times \mathbb{B})$.

$\mathbb{N}$ represents all possible role names that a meta association can have.

$K = \{None, Aggregate, Composite\}$ denotes the aggregation kind.

$\mathbb{B} = \{true, false\}$ determines whether the association is ordered or not.

Each meta association $e = (v_1, v_2) \in V \times V$ features two association ends. One end can have an aggregation kind different from $None$. This is depicted with a black or hollow diamond. The other association end can be ordered, depicted by the constraint `{ordered}`. By convention, the first association end (the one with the optional diamond) is called *left* meta association end. The other one (with the possible ordering constraint) is called *right* meta association end.

An example can be found in the UML meta model depicted in Figure 5.1. The meta association between `BehavioralFeature` and `Parameter` is a composition and it is ordered. This implies that the parameters are an integral part of the `BehavioralFeature` that owns them and the parameters of an `BehavioralFeature` are ordered.

Meta attributes (i.e. attributes of meta classes) have been separated from the above definition since they are not treated in a special way by the graph-based algorithms presented in this

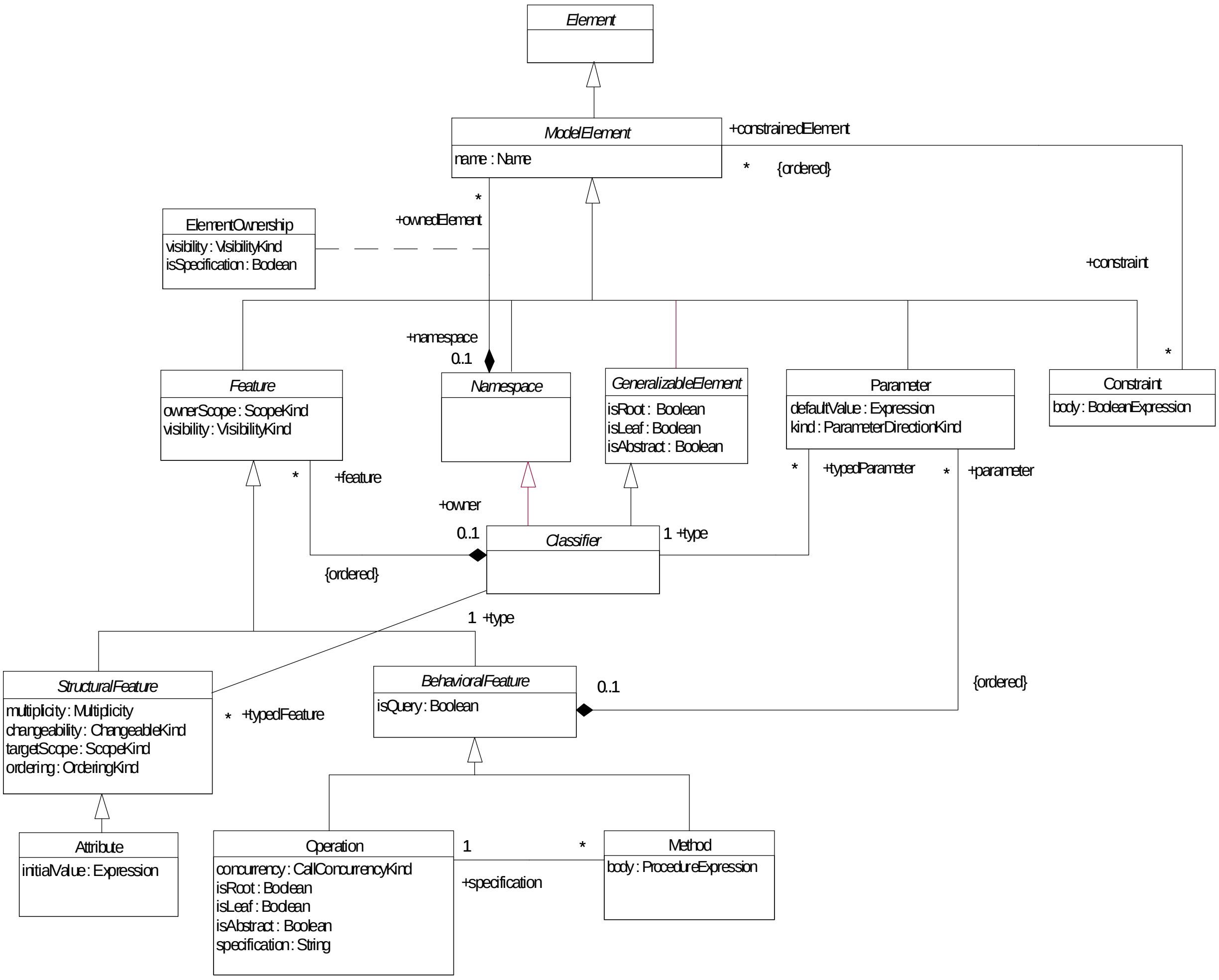


Figure 5.1: Core of the UML meta model

chapter. Meta attributes can be reduced to labels attached to vertices of the meta model. Hence, they are not important for graph algorithms. However, they are relevant for the Python binding discussed in section 8.3. The following definition adds meta attributes to each meta class.

Definition 5.2. A *meta attribute* of a meta model $\mathcal{M} = (V, E)$ is defined as an element of $\mathbb{N} \times V$. Here, $\mathbb{N}$ denotes the name of the attribute. Possible types of meta attributes are meta classes. Hence, V is the set of possible meta attribute types.

The function $a_{\mathcal{M}}(v)$ maps every vertex $v \in V$ (i.e. each meta class) to a set of meta attributes.

A full featured MOF meta model can have even more properties. For example, associations feature multiplicities and meta classes can have meta operations (see [OMG02d] for a more detailed explanation). Furthermore, enumerations are supported as a special kind of meta classes. However, a formal model should be as simple as possible. The additional properties can be mapped to labels and the above definitions are sufficient for the following discussion.

A model can be represented as a directed, typed, and labeled graph as shown by the following definition.

Definition 5.3. A *model* M based on the meta model $\mathcal{M} = (V_{\mathcal{M}}, E_{\mathcal{M}})$ is defined as a directed, typed, and labeled multi-graph $M = (V, E)$ with $E \subseteq V \times V$.

The graph is typed. That means every vertex in V and every edge in E corresponds to a vertex in $V_{\mathcal{M}}$ or $E_{\mathcal{M}}$, respectively. The functions $t_V : V \mapsto V_{\mathcal{M}}$ and $t_E : E \mapsto E_{\mathcal{M}}$ type the graph.

For every $e = (v_1, v_2) \in E$ and $\mathcal{T} = t_E(e)$ the vertex v_1 plays the role of the left association-end while v_2 is the right association-end. If $\mathcal{T}$ has an aggregation kind of *Composite*, v_1 is a composition and v_2 is a part. If $\mathcal{T}$ has an aggregation kind of *Aggregate*, v_1 is an aggregation and v_2 is a part.

The vertices are labeled. The label of a vertex x contains a value for each meta attribute in $a_{\mathcal{M}}(t_V(x))$.

The edges are not labeled.

5.2 Shortcomings of Standard Graph Transformation

Figure 4.3 shows that a graphical representation of transformation rules is possible. For convenience the relevant part of Figure 4.3 is repeated in Figure 5.2. This figure is based on the theory of graph-rewriting that is discussed in section 4.4.

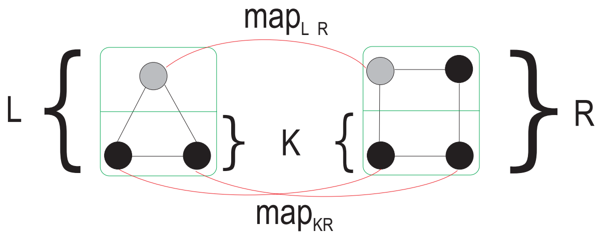


Figure 5.2: Graphical representation of a transformation rule.

Unfortunately, this notation cannot be applied directly to MDA-style model transformations. First, requirement 3.2 states that the standard UML notation should be applied whenever possible. Second, the theory behind this notation assumes that the start graph and the final derivation graph are subject to the same meta model and well-formedness rules. In fact, that is not the case for the MDA. The start graph (G) is subject to the PIM meta model while the final derivation graph (H) adheres to the PSM meta model. The derivation graphs between G and H are problematic.

$$G \Rightarrow_{r_1} G_1 \Rightarrow_{r_2} G_2 \Rightarrow_{r_3} \cdots \Rightarrow_{r_n} H$$

The graphs $G_i, 1 \leq i < n$ would be a mixture of PIM and PSM. Even merging both meta models into one intermediate meta model is no viable solution. This is revealed by Figures 5.2 and 4.3. In terms of the MDA, the graph L is based on the PIM meta model, just as the entire start graph G. The graph L will be replaced by R. The graph R in turn adheres to the PSM meta model, just as the entire final derivation graph H. The hitch is the sub-graph K of L, because it must have an occurrence in R, too. Obviously, graph K cannot have an occurrence in two graphs that are based on different meta models. A second problem is the reconnecting of edges as shown in Figure 4.3. Each edge adheres to a meta association. If one vertex is part of the PSM and another one of the PIM, there is obviously no meta association available that could serve this purpose. Hence, the glued graph R would remain isolated.

Another problem arises from requirement 3.4. It claims that the transformation process must support iterative development processes. As detailed in chapter 6, this requires some kind of mapping between the PIM and the derived PSM. Otherwise changes made to the PSM could not be re-integrated during a subsequent transformation. If the outcome of the transformation results just in a PSM model then the transformation cannot yield the required mapping between PIM and PSM, because the PIM has been discarded during the transformation.

5.3 Adaptation to the MDA

Two problems have to be solved in order to adapt graph-rewriting concepts to the MDA:

- the mix of meta models in the derivation graphs G_i, and
- the mapping between PIM and PSM to satisfy requirement 3.4.

The required mapping should be a direct outcome of the transformation process. After the transformation, labeled edges should connect elements in the PIM with their relatives in the PSM. Therefore, the PIM itself must be part of the final derivation graph. The final derivation graph is denoted as H throughout this chapter. Furthermore, the mapping itself will require a meta model of its own. This has implications on the meta model problematic outlined above.

The graph H must be based on a meta model that merges the PIM with the PSM and mapping meta model. In the following the PIM meta model is $\mathcal{G}$ and the PSM meta model is denoted as $\mathcal{S}$.

Definition 5.4. The *mapping meta model* $\mathcal{R}$ is defined as

$\mathcal{R} = (\{Ref\} \cup \mathrm{top}(V_G) \cup \mathrm{top}(V_S), E_R)$.

$\mathcal{G} = (V_G, E_G)$ is the PIM meta model and

$\mathcal{S} = (V_S, E_S)$ the PSM meta model.

top(V) is a set of meta classes of V which are not subclasses of other meta classes.

The edges of $\mathcal{R}$ connect all toplevel meta classes of $\mathcal{G}$ and $\mathcal{S}$ with Ref. Hence,

$E_R = \{e \mid e = (g, Ref), g \in top(V_G)\} \cup \{e \mid e = (Ref, s), s \in top(V_S)\}$.

With these definitions the merged meta model $\mathcal{H}$ can be defined as follows.

Definition 5.5. The *merged meta model* is defined as $\mathcal{H} = (V_G \cup V_S \cup \{Ref\}, E_G \cup E_S \cup E_R)$.

$\mathcal{G} = (V_G, E_G)$ is the PIM meta model,

$\mathcal{S} = (V_S, E_S)$ the PSM meta model, and

$\mathcal{R} = (V_R, E_R)$ the mapping meta model.

$V_R = \{Ref\}$.

If G is the PIM model and H is the result of the transformation and

$$G \Rightarrow_{r_1} G_1 \Rightarrow_{r_2} G_2 \Rightarrow_{r_3} \cdots \Rightarrow_{r_n} H$$

then the meta model for G is $\mathcal{G}$ and the meta model for $G_i, 1 \leq i < n$ and H is $\mathcal{H}$. From definition 5.5 follows that $\mathcal{G}$ is a sub-graph of $\mathcal{H}$. Hence, G adheres to the meta model $\mathcal{H}$, too. This means that the entire transformation process can be based on a single meta model: $\mathcal{H}$. This solves the meta model merging problem.

Every meta class of the $\mathcal{G}$ (PIM) is connected directly or via one of its super classes to the meta class R. R in turn is connected directly or indirectly via super classes to every meta class of $\mathcal{S}$ (PSM). Hence, $\mathcal{H}$ is able to describe a directed mapping between the PIM and the PSM. This solves the mapping problem.

This leads to a modified definition of a transformation rule (see definition 4.2).

Definition 5.6. A MDA transformation rule r is defined as

$r = (\mathcal{G}, \mathcal{S}, \mathcal{R}, L, R, K, map_{KR}, map_{LR}, appl)$.

$\mathcal{G}$ is the PIM meta model,

$\mathcal{S}$ is the PSM meta model, and

$\mathcal{R}$ is the mapping meta model. $\mathcal{H}$ is the merged meta model of $\mathcal{G}$, $\mathcal{S}$, and $\mathcal{R}$.

The graphs L and R adhere to $\mathcal{H}$.

All vertices of K are instances of the meta classes of $\mathcal{G}$. K is a sub-graph of L and has an occurrence in R as defined by map_{KR}. This mapping is type preserving. Hence, if $v \in K$ is of meta class M, the vertex $map_{KR}(v) \in R$ adheres to the same meta class.

The mapping map_{LR} maps some (or none) vertices of L to R. This mapping is type preserving, too. Hence, if $v \in L$ is of meta class M, the vertex $map_{LR}(v) \in R$ adheres to the same meta class.

Every vertex in L and R that is an instance of a meta class of $\mathcal{G}$ is covered by K. Every edge in L and R that is an instance of a meta association of $\mathcal{G}$ is covered by K, too.

appl is, as usual, an application constraint.

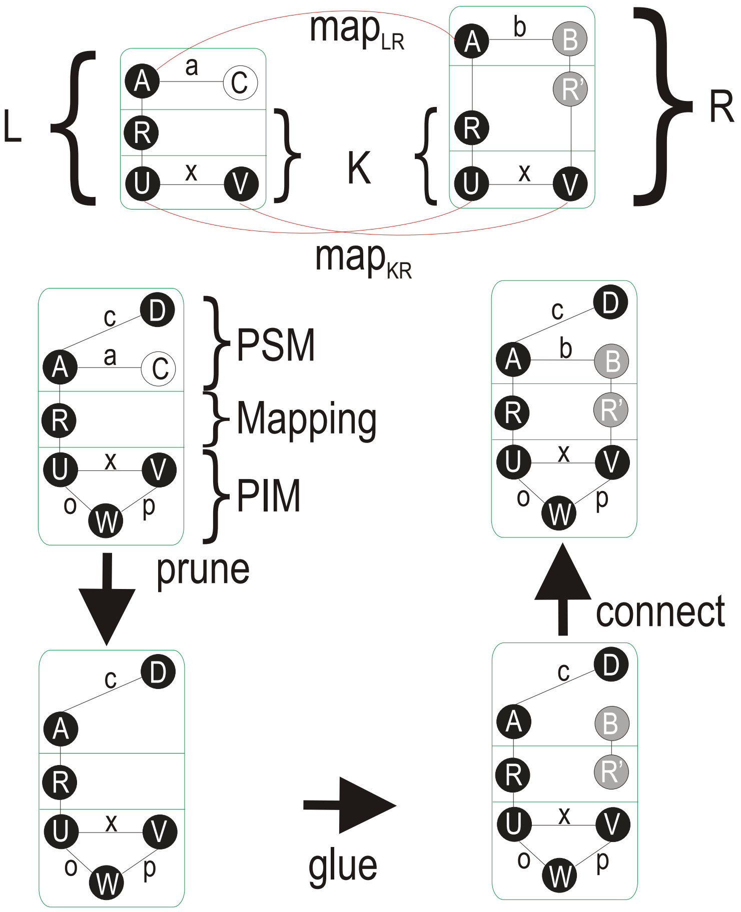


Figure 5.3: Example of a MDA transformation rule and the transformation process.

Figure 5.3 provides an example of an MDA transformation rule and the transformation process. The two graphs on top represent the transformation rule consisting of L, R, K, map_{KR}, and map_{LR}. The three meta models $\mathcal{G}$, $\mathcal{S}$, and $\mathcal{R}$ are not shown. Nodes are labeled with capital letters while edges are labeled with lower case letters. These labels correspond to meta classes and meta associations of the respective meta models. The two graphs are split in three sections. The top most is reserved for elements of the PSM ($\mathcal{S}$), the PIM ($\mathcal{G}$) at the bottom and the mapping ($\mathcal{R}$) in between. The sub-graph K covers all element adhering to $\mathcal{G}$ as demanded by definition 5.6. It is worth noting that all edges crossing the section borders are connected to an element of $\mathcal{R}$ since $\mathcal{R}$ glues the two other meta models together. The two mappings map_{KR} and map_{LR} have the same meaning as in Figure 4.3. The purpose of the transformation rule is to remove node `C`, create node `B` and extend the mapping with `R'`. Different fillings are used in the figure to illustrate this.

The start graph is shown below the transformation rule. The first step is to prune the graph. Hence, `C` is removed. Then, the new nodes `B` and `R'` are added. Finally, the missing edges are inserted.

The important difference between definition 5.6 and definition 4.2 is the way L, R, and K are constrained to the different meta models. Note that $\mathcal{R}$ can be used by L and R. Hence, the mapping between PIM and PSM can be created throughout the transformation and it can be used for pattern matching. Furthermore, the mappings map_{LR} and map_{KR} are type preserving. This was not demanded by the previous definition. Another important property of an MDA transformation rule is that it preserves the PIM. This is not directly obvious from the definition. A proof is necessary.

Proposition 5.1. If r_i are MDA transformation rules and graph G_0 represents a PIM and adheres to $\mathcal{G}$ then all derivations preserve the start graph G_0.

Hence, from $G_0 \Rightarrow_{r_1} G_1 \Rightarrow_{r_2} \cdots \Rightarrow_{r_n} H$ follows that G_0 is a sub-graph of H.

Proof. The r_i are defined as $(\mathcal{G}, \mathcal{S}, \mathcal{R}, L, R, K, map_{KR}, map_{LR}, appl)$. The structure of K is preserved throughout a transformation. This follows from observation 4.1. The types of the vertices and edges of K are preserved because map_{KR} is type preserving.

$\Rightarrow$ K remains unchanged by a transformation.

From definition 5.6 follows that K covers all vertices and edges of L that adhere to $\mathcal{G}$.

$\Rightarrow$ All vertices and edges of L that adhere to $\mathcal{G}$ remain unchanged.

Throughout the application $G_{i-1} \Rightarrow_i G_i$ only those elements that are matched by L can be changed.

$\Rightarrow$ The PIM elements in G_{i-1} are either not matched by L or covered by K.

$\Rightarrow$ The PIM elements remain unchanged.

$\Rightarrow$ The proposition is true because G_0 consists of PIM elements only. $\square$

This leads to another important property of MDA transformations.

Proposition 5.2. The graphs L and R share all vertices and edges that adhere to the PIM meta model ($\mathcal{G}$).

Proof. $r = (\mathcal{G}, \mathcal{S}, \mathcal{R}, L, R, K, map_{KR}, map_{LR}, appl)$ is a transformation rule.

All vertices and edges in L that adhere to the meta model $\mathcal{G}$ are by definition 5.6 covered by K. The same argument holds for R and K.

K is a sub-graph in L and has an occurrence in R as demanded by definition 5.6.

Furthermore, map_{KR} is type preserving.

$\Rightarrow$ the mapping map_{KR} is bijective and preserves structure and type of the mapped graph.

$\Rightarrow$ If a vertex or edge adheres to $\mathcal{G}$ and is part of either L or R then the bijection maps it to R or L respectively. ☐

This proposition is useful to simplify the notation as the following section will show.

5.3.1 Notation

Figure 5.3 shows that it is possible to provide a graphical notation for transformation rules. However, the notation can be significantly simplified.

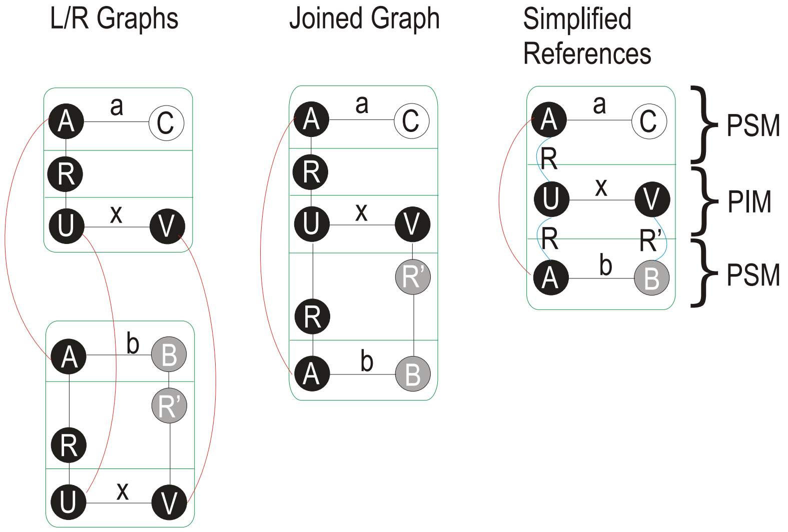


Figure 5.4: Simplification steps

Figure 5.4 illustrates two simplification steps. The graph on the left hand side is the same as the one in Figure 5.3. The graphs L and R share the same PIM section. This is always the case as shown by proposition 5.2. Hence, it is better to share the visual representation of K. This is realized by the graph in the middle. Since K is visible only once, the mapping map_{KR} can be dropped from the notation, too. The second step is to simplify the part of the graph that is based on $\mathcal{R}$. The meta model $\mathcal{R}$ adds only the Ref meta class to $\mathcal{H}$. Instead of depicting instances of Ref as a circle it can be depicted as a labeled line that connects a PIM vertex with a PSM vertex. This simplification is shown on the right hand side.

The last redundancy in the graph is the double occurrence of the line labeled `R`. This line denotes an instance of the Ref meta class of meta model $\mathcal{R}$. The meaning of the top occurrence is that `R` is part of the search graph L. The occurrence below denotes that `R` is not changed by the replace graph R. Hence, `R` is preserved by the transformation. Experience has shown that MDA graph transformations will usually not remove instances of the Ref meta class. Usually, they are added only by each application of a transformation rule. By demanding that instances of the Ref meta class are always preserved, the bottom occurrence of the line can be discarded, too.

5.3.2 Chaining Transformation Rules

A complete transformation of a PIM into a PSM is not possible with only one rule. A set of rules has to be applied in a certain order. Experience has shown that many transformation rules depend on each other. As shown in the following example, this leads to redundancy among the transformation rules.

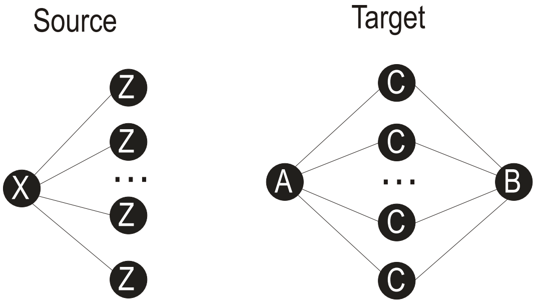


Figure 5.5: Source and target graph of a transformation requiring two rules

Figure 5.5 shows a class of source graphs and a class of target graphs. Instances of this class differ in the number of `Z` or `C` vertices. To transform each graph of the source class into its counterpart of the target class two transformation rules are sufficient. However, it is impossible to handle the problem with one rule only.

Figure 5.6 shows two transformation rules that solve the problem. The hitch is that the `A` and `B` vertices and the mapping labeled with `R` and `R'` appear three times. This is redundancy. Hence, the well known implications of redundancy will show up. If `Rule0` changes during development of a transformation, no tool can detect that `Rule1` should be adapted. Furthermore, `Rule1` looks more complicated then it should.

Another drawback is the performance of the transformation process. Transformations based on graph rewriting tend to be slow, because the problem of sub-graph isomorphism is NP-complete

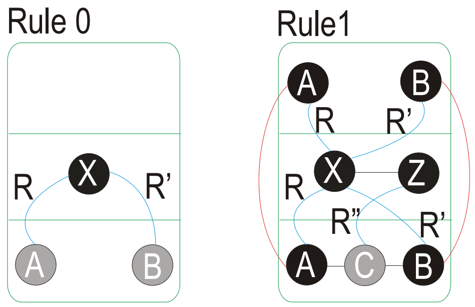


Figure 5.6: Implicitly chained transformation rules

[GJ90]. For small graphs L and typical transformations the average runtime is not exponential. However, it is far from being linear. Chained rules as shown in Figure 5.6 can be optimized because `Rule1` searches for all occurrences of a pattern that has been created by `Rule0`. A possible optimization would take that into account. Hence, the application of `Rule1` would not require any sub-graph search.

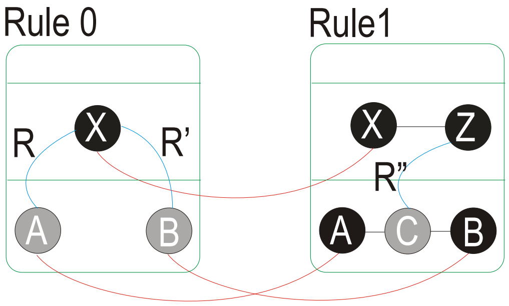


Figure 5.7: Explicitly chained transformation rules

Both problems, the redundancy and the optimization, can be addressed by explicitly chaining rules. This is shown in Figure 5.7. Two lines connect `Rule0` with `Rule1`. They represent a mapping that relates the `A`, `B`, and `X` vertices of `Rule0` with the `A`, `B`, and `X` vertices in `Rule1`.

With this mapping, the optimization can be easily achieved. `Rule1` is applied whenever `Rule0` has been applied. Just the `Z` nodes have to be searched. That is much more efficient than searching for every occurrence of the four vertices `A`, `B`, `X`, and `Z`. Furthermore, `Rule1` has reduced the redundant information to the possible minimum. The purpose of the rule is easier to understand and relationship between rules has been made explicit. Chaining of transformation rules provides no advance in computational power. It is a very useful optimization with respect to readability and performance.

Chapter 6

Round-Trip Engineering

Modern development processes [DW98, JBR99, Bec99, Agi02] are iterative. This is in contrast to the waterfall processes. In waterfall processes the development consists of consecutive steps. No step is taken twice and there is no stepping back. Iterative processes instead repeat the sequence of steps until the product is finished. In practice this means that a certain set of requirements is analyzed, implemented and tested. Then the same procedure is repeated with additional requirements.

The problem of combining the MDA with iterative processes is located in the tools. All generative approaches provide means for generating an artifact B out of another artifact A. In the case of the MDA such artifacts are models. However, an artifact may be some kind of text, graph, etc. During each iteration B is altered by the developers and the result is an artifact $\tilde{B}$. When the next iteration runs, developers modify A resulting in A'. The generation process derives B' from A'. This yields the problem that the changes between B and $\tilde{B}$ are lost. There are two possibilities of circumventing this problem. The first one is to realize the round-trip engineering via *reverse engineering*. Here $\tilde{B}$ is reverse engineered to yield $A_{reverse}$. Hence, the differences between B and $\tilde{B}$ are detected, translated and applied to A. This allows starting the next iteration without any loss of information. The second possibility is to automatically apply the differences between B and $\tilde{B}$ to B' yielding $\tilde{B}'$.

6.1 Reverse Engineering

Both presented solutions have advantages and disadvantages. It is the goal of this section to show that reverse engineering is in general not an appropriate solution in the context of the MDA.

Reverse engineering is still a real tooling problem for the source code generation. Source code is generated from a UML model. This code is modified by developers. Reverse engineering re-translates the code into UML models. It does not come by surprise that this works pretty well for class diagrams. The reason is that class diagrams and object-oriented languages share a large set of concepts. Hence, forward engineering (generation) and reverse engineering are comparably easy because of an isomorphism between class diagrams and class definitions in OO programming languages.

The case is different for state charts, sequence diagrams or activity diagrams. Commercial tools often fail to generate code from these diagrams. The reverse is almost impossible. The reason is that the concepts used to describe behavior in the UML are quite different from the more low-level constructs of programming languages. This leads to the following observation.

Observation 6.1. A generation tool reads an artifact A based on the set of concepts $\mathcal{A}$. The output is an artifact B based on the concepts $\mathcal{B}$.

Reverse engineering has proven to be possible if there exists an isomorphism between the concepts in $\mathcal{A}$ and $\mathcal{B}$, hence, $\mathcal{A} \equiv \mathcal{B}$.

Reverse engineering has proven to be difficult if $\mathcal{A}$ and $\mathcal{B}$ differ in the way they describe things even if both sets of concepts have the same descriptive power.

Another problem is that one round-trip must not loose information. In the above example $\tilde{B}$ is reverse engineered into $A_{reverse}$. If forward engineering is applied to $A_{reverse}$ then it should generate $\tilde{B}$ again. Otherwise, the reverse engineering would loose information. Furthermore, forward engineering must work loss-less. Otherwise B would contain less information than A, hence, information would be lost. This leads to the next observation.

Observation 6.2. The function f implements forward engineering and reverse engineering is represented as the function r.

$\Rightarrow$ The functions $f \circ r$ and $r \circ f$ are the identity function.

Forward engineering and reverse engineering translate information between two different sets of concepts: $\mathcal{A}$ and $\mathcal{B}$. From the fact that forward engineering works loss-less follows that $\mathcal{B}$ has at least the same descriptive power as $\mathcal{A}$. Developers may change the output of the forward engineering. Hence, they may utilize every concept in $\mathcal{B}$. Since reverse engineering works loss-less, too, it follows that $\mathcal{A}$ has at least the same descriptive power as $\mathcal{B}$. The following observation can be deduced from this.

Observation 6.3. The function f projects $\mathcal{A}$ onto $\mathcal{B}$.

The function r projects $\mathcal{B}$ onto $\mathcal{A}$.

f and r are forward engineering respectively reverse engineering functions in the sense of observation 6.2.

Then the set of concepts $\mathcal{A}$ has the same descriptive power as $\mathcal{B}$, hence, $\mathcal{A} \equiv \mathcal{B}$.

Reverse engineering is in general not the appropriate solution for the MDA. The reason why the MDA has been chosen as the fundamental approach for this thesis is that the problem domain for QoS-enabled applications is quite different from the realization domain. There is no isomorphism between the concepts used in the problem domain and those in the realization domain. Following observation 6.1, this renders reverse engineering especially complicated.

It is obvious that the PIM meta model presented in this thesis has less descriptive power than the UML. Hence, observation 6.3 does not hold for the PIM and PSM meta model. Therefore, developers would have to work with a loss of information during reverse engineering.

6.2 Forward Merging

Pure reverse engineering adheres to the idea that a single artifact contains all information about the system at a time. In the case of the MDA such an artifact is a model. Another approach is to accept that all knowledge about a system is distributed across two models: the PIM and the PSM. In an iterative process changes are applied to both models. Therefore, a synchronization mechanism is required. Forward merging provides such a mechanism.

Definition 6.1. I_n is a list of PIM models. S_n and $\tilde{S}_n$ are lists of PSM models. During iteration j the PSM model S_j is generated from I_j via model transformation. Developers will modify S_j yielding $\tilde{S}_j$. Throughout the next iteration S_{j+1} will be generated.

Forward merging automatically merges the difference between S_j and $\tilde{S}_j$ into S_{j+1}. Hence, S_{j+1} is a synchronization between the PIM model I_{j+1} and the latest PSM model $\tilde{S}_j$ touched by the developer.

The following observation shows that forward merging overcomes a central problem of reverse engineering.

Observation 6.4. Forward merging can be expressed as a function r that projects a set of concepts $\mathcal{A}$ onto another set of concepts $\mathcal{B}$.

It is *not* required that the expressive power of $\mathcal{A}$ is equivalent to that of $\mathcal{B}$. This follows from the fact that the information is distributed among two models.

However, it could happen that some PSM model $\tilde{S}_k$ contradicts the PIM model I_{k+1}. This happens if developers apply substantial changes to S_k resulting in $\tilde{S}_k$ and they do not adapt I_{k+1} to reflect these changes, too. The next model transformation cannot succeed because of this contradiction. This problem arises from the fact that all information about the system is distributed across two models. The two models can diverge and contradict each other.

The mismatch between $\tilde{S}_k$ and I_{k+1} can be classified in *syntactic conflicts* and *semantic conflicts*. Syntactic conflicts appear, for example, if $\tilde{S}_k$ adds a new method to a class that exists in S_k, too, but I_{k+1} is modified in such a way that the generated PSM S_{k+1} does no longer contain this class. In this case the new attribute cannot be applied to S_{k+1} because its class is gone. Semantic conflicts can appear in behavioral diagrams. For example, $\tilde{S}_k$ adds transitions to a state machine that are fired when an event of type A is received. I_{k+1} is modified in such a way that the generated PSM S_{k+1} features transitions in the same state machine that fire when an event of type B is received. The state machine is semantically still correct. However, if event type B is a subtype of A, the modification applied to $\tilde{S}_k$ results in a different behavior when applied to S_{k+1}. In $\tilde{S}_k$ the transition fired if an event of type A was received, including events of the subtype B. In S_{k+1} the transition fires only if an event of type A is received that is *not* of subtype B. Syntactic conflicts can be detected while applying the differences between S_k and $\tilde{S}_k$ to S_{k_1}. Semantic conflicts are more difficult to detect. A deep analysis of the behavioral diagram would be required. To perform such an analysis on a UML-based PSM it has to be mapped to a more formal model such as Petri Nets which are better suited for formal reasoning.

The tools built throughout the work on this thesis can detect syntactic conflicts. Detected conflicts are reported to the developer. However, the changes between S_k and $\tilde{S}_k$ are always applied if anyhow possible. The model transformation does not fail if a conflict is detected. In case of a doubt the information stored in I_{k+1} precedes $\tilde{S}_k$. This may result in a loss of information. If this occurs then it can be pointed out by a tool and the developers have to investigate how to proceed. This loss of information is different from the one discussed in the section 6.1. Here, loss of information occures only in case of a conflict. An information-loss during reverse engineering would be systematic, hence, not a result of an error. Information would be lost during every iteration.

6.3 Forward Merging as a Graph Transformation

Forward merging has been selected as the mechanism of choice for the tool chain developed throughout this thesis. Some of the reasons and alternatives have been discussed in the previous sections. Another reason is that forward merging on models can be implemented independently of a concrete model transformation. This section shows how this can be achieved.

The independence of a concrete transformation is very important in the context of this thesis. Model transformation is subject to continuous extensions for new QoS categories or QoS implementations. Because of this independence there is no need to adapt the forward merging mechanism for every modification of the MDA model transformation.

Forward merging imposes two key challenges. First, the difference between S_j and $\tilde{S}_j$ has to be determined. Second, this difference has to be applied to the outcome of the next model transformation.

The developers modify S_j resulting in $\tilde{S}_j$. This modification can be treated as a model transformation. Hence, a transformation rule r (see definition 4.2) exists that derives $\tilde{S}_j$ from S_j.

$$S_j \Rightarrow_r \tilde{S}_j$$

The basic idea is to apply this transformation to the output of the model transformation. During iteration $j+1$ some MDA transformation m (see definition 5.6) derives the model X_{j+1} from I_{j+1}. X_{j+1} is just a temporary model. Applying r to X_{j+1} yields S_{j+1}.

$$I_{j+1} \Rightarrow_m X_{j+1} \Rightarrow_r S_{j+1}$$

The problem is that the transformation r is not known. Furthermore, the fact that r derives $\tilde{S}_j$ from S_j does not necessarily imply that it will add the difference between S_j and $\tilde{S}_j$ to X_{j+1}. However, the following observation highlights the important fact.

Observation 6.5. If r can be determined and

if it merges the differences between S_j and $\tilde{S}_j$ into X_{j+1} with $I_{j+1} \Rightarrow_m X_{j+1}$

then the transformation $m \circ r$ implements forward merging as defined in definition 6.1.

6.3.1 Deduction of the rule r

The model transformation r has to be derived somehow. One option is to follow step by step the modifications the developer applied to the model S_j. However, that would impose a major struggle on the UML tool.

The better idea is to extract r by inspecting S_j and $\tilde{S}_j$. The differences between two models can be categorized as follows.

- Vertices:
  - added
  - removed
- Edges:
  - added
  - removed
- Attributes (i.e. labels) changed

The UML-tool Kase (see chapter 8) uses unique IDs internally to mark vertices. Hence, vertices that are neither added nor removed share the same IDs in both models, i.e. S_j and $\tilde{S}_j$. Due to these IDs it is possible to efficiently detect the differences between S_j and $\tilde{S}_j$. This is important because otherwise the NP-completeness of graph isomorphism would render the entire approach into a very slow one.

The transformation r is defined as

$$r = (L, R, K, map_{LR}, map_{KR}, appl)$$

The sub-graph K is by definition not altered during the model transformation. Hence, as a first approximation K consists of all vertices that exist in S_j and in $\tilde{S}_j$. Furthermore, K covers all edges between its vertices that exist in S_j and in $\tilde{S}_j$, too.

The sub-graph K has an occurrence map_{KR} in R. The replace pattern R covers all vertices and edges in $\tilde{S}_j$ that either belong to K, are not present in S_j or have different attributes in S_j. The search pattern L covers all vertices in S_j that either belong to K, are not present in $\tilde{S}_j$ or have different attributes in S_j. If r covers vertices in $\tilde{S}_j$ that exist in S_j, too, then they are mapped to their corresponding vertices in L via map_{LR}. This leads to the following observation.

Observation 6.6. Given two models S_j and $\tilde{S}_j$ it is possible to deduce a model transformation r with the property

$$S_j \Rightarrow_r \tilde{S}_j$$

if vertices shared by both models have the same ID.

Another observation is that L covers S_j, because every vertex and edge in S_j will either exist in $\tilde{S}_j$ or it will be removed. Hence, r can be applied only on graphs that have an occurrence of S_j. However, that is not very beneficial in the light of observation 6.5, because it would be unlikely that r can be applied to X_{j+1}.

The transformation r can be simplified by removing vertices and thus edges from K. If a vertex in K is only connected with other vertices in K then it can be discarded. Obviously the property $S_j \Rightarrow_r \tilde{S}_j$ is not affected because the vertices covered by K are not modified by the transformation anyway. Furthermore, it is possible to remove all edges from K. These edges are not modified by the transformation.

The sole purpose of the removed vertices and edges is to find the occurrence of L - which includes K - in S_j. While the simplified L will still find at least one match in S_j, it could happen that it has more than one possible match. This would violate the property $S_j \Rightarrow_r \tilde{S}_j$. To overcome such problems, an ID is used that uniquely identifies each node. Then a match of L is only possible if the matched vertices have the required ID. Hence, the simplified transformation will have the required property $S_j \Rightarrow_r \tilde{S}_j$, too.

The result of the simplification is that L does not necessarily cover S_j completely since vertices and edges have been pruned in K. In the extreme case of S_j being equal to $\tilde{S}_j$ the transformation r would feature empty sub-graphs L, K, and R. In general, the complexity of the transformation increases as the differences between S_j and $\tilde{S}_j$ increase. Thus, the smaller the differences are the more likely is a successful application of r to X_{j+1}.

6.3.2 Generating Unique IDs

Applying the transformation r to X_{j+1} needs special precautions. The IDs mentioned above are a problem, since X_{j+1} is generated via a MDA model transformation that is not aware of forward merging. The vertices in X_{j+1} have completely different IDs than those in S_j and $\tilde{S}_j$. Thus, the transformation r could not be applied to X_{j+1}.

The IDs of S_j and X_{j+1} differ completely. Those of I_j and I_{j+1} stay the same. Since the transformation m is a MDA model transformation there exists a mapping between vertices of I_j and S_j based on the mapping meta model $\mathcal{R}$. Instead of using the ID of a vertex in S_j, the ID of the corresponding vertice(s) in I_j can be used. Furthermore, the vertices that are instances of the mapping meta class $Ref \in V_{\mathcal{R}}$ are labeled. This label is included in the ID, too.

Definition 6.2. $I = (V_I, E_I)$ is a PIM and $S = (V_S, E_S)$ is a PSM that has been derived from I. The mapping graph $R = (V_I \,\cup\, V_S \,\cup\, V_R,\; E_R)$ is structured as follows:

$\forall r \in V_R \;\exists\; (v, r) \in E_R$ with $v \in V_I$.

$\forall r \in V_R \;\exists\; (r, v) \in E_R$ with $v \in V_S$.

Hence, every $r \in V_R$ has exactly one outgoing and one incoming edge.

Then the *ID* of a vertex $s \in V_S$ is defined as the set

$ID(s) = \{(v, \ell) \;\mid\; \exists\; (v, r), (r, s) \in E_R \;\wedge\; \ell$ is the label of $r\}$.

Following this definition, a vertex s_1 of S_j and a vertex s_2 of S_{j+1} correspond if $ID(s_1) = ID(s_2)$. The most important property of the ID function is that the computed set does not reference the PSM any more. The resulting ID builds entirely on the PIM and labels on edges of V_R. Furthermore, it is important to notice that the mapping graph connects *every* vertex in V_S via at least one vertex $r \in V_R$ with a vertex in V_I. Otherwise, the ID function could return an empty set for some vertices of V_S.

The basic idea of the ID generation is that the ID describes *why* an element in S_j has been created. If an MDA model transformation is repeated, the generated elements will always have the same ID if they have been created for the same reason. To achieve uniqueness of this ID, it is important that the labeling of the mapping graph is chosen cleverly. Chapter 7 explains how the mapping graph and its labels can be chosen to guarantee unique IDs.

6.3.3 Application of the rule r

The rule r that applies the changes between S_j and $\tilde{S}_j$ to X_{j+1} yielding S_{j+1} requires a special application algorithm. It could happen that some vertices that have been present in S_j are no longer present in X_{j+1}. If this vertex is present in the search pattern L of r then r cannot be applied to X_{j+1} because the vertex is missing.

This problem can be tackled with the help of fuzzy sub-graph matching and the spanning tree of the model. If a vertex of L could not be mapped to a vertex in X_{j+1}, such a vertex is introduced temporarily in X_{j+1} and marked with a flag. Thus, L will always find an occurrence in X_{j+1} and the transformation can take place. After the transformation the spanning tree over the generated model S_{j+1} is determined. For every vertex in S_{j+1} that has been marked with a flag previously the entire sub-tree with regard to the spanning tree is deleted. Hence, the temporarily inserted vertices are removed together with all of their owned elements.

An example illustrates this idea. The PIM model I_j contains a set of components. These are translated into classes and state charts in the PSM model S_j. Finally, the developers modify S_j resulting in $\tilde{S}_j$. During the next iteration one component is removed from the PIM, hence, I_{j+1} misses a component. The PSM X_{j+1} generated from I_{j+1} misses the corresponding class and state chart. Hence, to apply the transformation r the class and its state chart are temporarily added to the model as dummies. After the application of r these dummies and all additional elements added to them during the transformation are removed.

The hitch of this approach is that the resulting model S_{j+1} may have broken semantics since the way r has been applied is rather crude. Thus, the UML tool should tell the developer that certain elements of $\tilde{S}_j$ could not be re-applied. Then the developer has to take care of the problem. Nevertheless, the algorithm tried to re-apply the differences between S_j and $\tilde{S}_j$ as good as possible.

Neither reverse engineering nor forward merging provide a 100% solution for the round-trip engineering problem. Vanilla reverse engineering has the problem that loss-less transformation from PSM to PIM is usually impossible. Forward merging has an inherent danger of conflicts. However, the same applies to all source code version management systems such as CVS [CVS03], Subversion [Sub03], or Perforce [Per03]. If multiple actors modify the same system independently, synchronization may end up in conflicts. Nevertheless, version management became an integral tool for developer teams despite this problem.

6.4 Preserving Diagrams

Round-trip-engineering should not loose information. Such information is not only located in the models themselves. Developers invest a significant amount of time in arranging the elements visually in diagrams. If the model transformation m that transforms I_{j+1} into X_{j+1} would invalidate all diagrams then the entire approach is questionable. No efficient algorithms for generating “nice” diagrams are known. For most kinds of diagrams optimizations of their layout is NP-complete. Hence, the goal is to display the new model S_{j+1} using the diagrams of the old model $\tilde{S}_j$. However, it may happen that S_{j+1} lost some of the vertices present in

$\tilde{S}_j$. Furthermore, S_{j+1} can contain new additional vertices. Thus, the old diagrams have to be automatically adapted to the new model S_{j+1}.

Diagrams can be seen as an instance of a special meta model. In fact UML 2.0 will contain a specification that describes a meta model for diagrams [OMG03h]. Such a diagram would be directly linked to the model. Since the diagram meta model follows definition 5.1 it is possible to determine the spanning tree of a diagram. If a vertex that has a representation in the diagram is missing in S_{j+1} then its representation is removed from the diagram. Due to the spanning tree all elements visually hosted inside the removed graphical element can be removed, too.

If a new vertex is present in S_{j+1} then the diagram should show it automatically. Therefore, Kase's event bus can be used. Whenever a new element shows up in the model, an event is generated and Kase tries to adapt the diagram. Hence, every element present in S_{j+1} but not in $\tilde{S}_j$ will cause an event to be sent. This event in turn will trigger Kase's diagram objects and they will try to reflect the change propagated by the event.

Chapter 7

Kafka

The previous chapter discussed the pros and cons of imperative and rule-based model transformation languages. Kafka [WUG03a] is a transformation language that unites advantages of rule-based and imperative approaches. Kafka is based on graph transformation techniques. A Kafka transformation rule consists of an orchestrated set of graph transformation rules. Each rule may be enriched with Python expressions. Graph transformation is useful to change the structure of a graph. This graph is a model (see definition 5.3), hence, its edges and vertices are labeled. In contrast, an imperative programming language such as Python is very well suited to work on the labels attached to the graph vertices.

One of the most important goals of Kafka is to enable developers who are not experts in the PIM and PSM meta model to implement model transformation. Therefore, Kafka is a visual transformation language. Its notation is based on the notation of the PIM and PSM.

7.1 Notation

The notation of transformation rules as shown in Figure 7.1 is not very intuitive. Developers need to understand the meta model of the PIM and PSM to understand what such a transformation rule does. The problem is that instances of all meta classes are depicted in the same way: the name of the meta class (or a shortcut thereof) is put inside a circle. For example, an instance of the `Realization` meta class is depicted just like the instance of the `Interface` meta class. Instead, most modeling languages provide a special graphical notation for instances of certain meta classes. A `Realization` is depicted as a dashed line with an arrow and an interface is depicted as a box labeled with the string ≪`interface`≫.

Requirement 3.2 demands that the notation of a transformation rule should be based on the notation of the PIM and PSM. Thus, the graph structure shown in Figure 7.1 should be replaced by the respective PIM and PSM notation. The result is shown in Figure 7.2. Both transformation rules share the same semantics but their notation is different. Only the idea of drawing a rule as a box with three compartments has been retained. The first compartment contains elements of the PSM that are part of the search graph L (see definition 4.2) and is called *PSM search pattern*. The compartment at the bottom is called *PSM replace pattern* and contains elements of the PSM, too. They replace the elements of the PSM search pattern when the rule is applied. Hence, these elements are part of the graph R. The compartment in the middle contains the *PIM search pattern*. The PIM is invariant throughout the entire transformation. Hence, the elements in this compartment are part of the graph K.

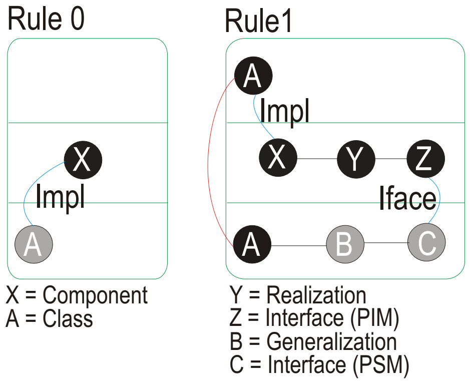


Figure 7.1: Transformation rules using graph-style notation

The mapping map_{LR} follows from the fact that that two model elements (one in the PSM search pattern, the other in the PSM replace pattern) share the same name. For example, in the right rule in Figure 7.2 two classes share the same name. This implies that the element from the PSM search pattern (graph L) is mapped to the element of the PSM replace pattern (graph K) via map_{LR}. If an element in the PIM search pattern and one in the PSM replace pattern are equally named, the name attribute of the PSM element will be set to the name of the PIM element. This is just for convenience because names are usually retained during model transformations. K is represented by the PIM search pattern. The mapping map_{KR} is not required since K has already been isolated from L (the PSM search pattern) and R (the PSM replace pattern). This simplification has been discussed in subsection 5.3.1.

The mapping graph needs a graphical notation. In Figure 7.1, the edges of the mapping graph are depicted as labeled lines crossing the border of the compartments. Figure 7.2 renders the edges of the mapping graph as dashed lines with an open arrow. The labels are depicted next to these lines.

Finally, the chaining of transformation rules as introduced in subsection 5.3.2 requires a notation. Drawing lines between different transformation rules as depicted in Figure 7.3 is no viable solution. Different rules may be shown on different diagrams. In this case the lines would not be visible. Therefore, the idea is to utilize the name. If an element of one rule wants to refer to an element of another rule, the name of the other rule is prefixed. This is illustrated by Figures 7.3 and 7.4. In Figure 7.4, `Rule1` references two times an element called `Rule0::Component`. References made in the PIM search pattern always refer to an element in the PIM search pattern of the referenced rule. The same principle applies to the PSM replace pattern. Its elements must only reference elements in another PSM replace pattern.

References between elements in two PSM search patterns are not allowed. References between a PSM replace pattern and a PSM search pattern are allowed. Elements in the PSM search pattern are part of the graph L. Either they have a corresponding element in graph R defined by the mapping map_{LR} or not. If not, this element is deleted by this rule. It would not make sense to refer to an element that is deleted before the referencing rule is executed. If the element is not deleted, it has a correspondence in graph R. Hence, there is no drawback in prohibiting references between two PSM search patterns.

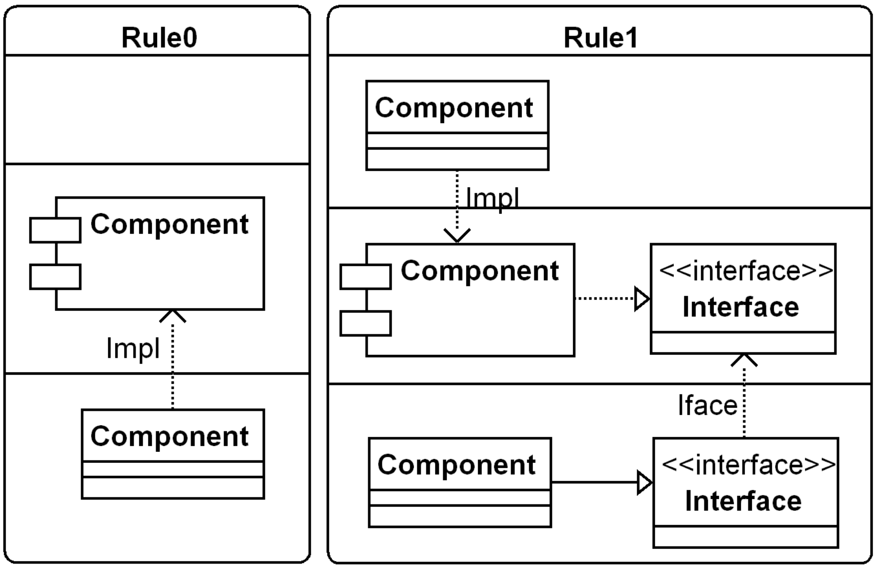


Figure 7.2: Transformation rules using UML-style notation

7.2 Object Diagrams

The notation of a Kafka rule is always based on the notation of the PIM and PSM. This is a great advantage because rules are easy to read if you understand the PIM and PSM notation. In some rare cases this approach can cause more problems than it actually solves, especially if the rule developer has detailed knowledge of the meta model. For such experienced developers Kafka features a special notation for rules that is closer to the PIM and PSM meta model. One or multiple patterns of a rule can be notated as object diagrams. In UML, object diagrams are used to depict instances of a model, i.e. objects of the M0 meta level. In Kafka, object diagrams are used to depict instances of a meta model, i.e objects of the M1 meta level. Figure 7.5 shows two equivalent Kafka transformation rules. Only their notation is different. In the rule on the right the two PSM patterns are depicted as a graph. The vertices are instances of meta model classes and the edges are instances of associations on the meta model level. This is comparable to the notation used in Figure 7.1. The major difference is that object diagrams use UML standard notation while Figure 7.1 uses a notation that is more common in graph transformation theory.

Figure 7.6 provides another example. Both rules change in every sequence diagram every invocation of a function `foo()` into an invocation of `bar()`. The rule depicted on the left-hand side of Figure 7.6 solves this task using the PSM notation. The rule on the right-hand side performs almost the same task using object diagrams. The difference between both rules is that the one using object diagrams changes synchronous and asynchronous invocations while the other one changes only synchronous invocations. The reason is that the rule on the left-hand side shows a synchronous invocation. Therefore, the search pattern will not match asynchronous invocations. The rule on the right-hand side does not make any assumption about the `isConcurrent` attribute of the `action` object. Hence, it matches both invocation modes. This illustrates that object diagrams in Kafka rules can save some work if the developers know

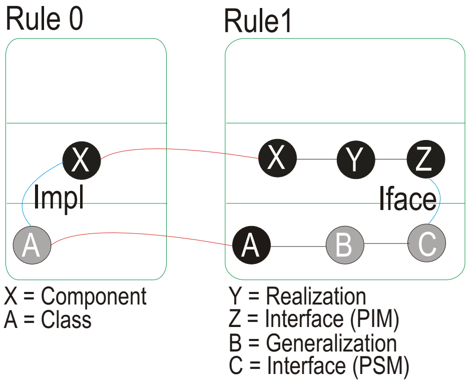


Figure 7.3: Chained transformation rules without UML-style

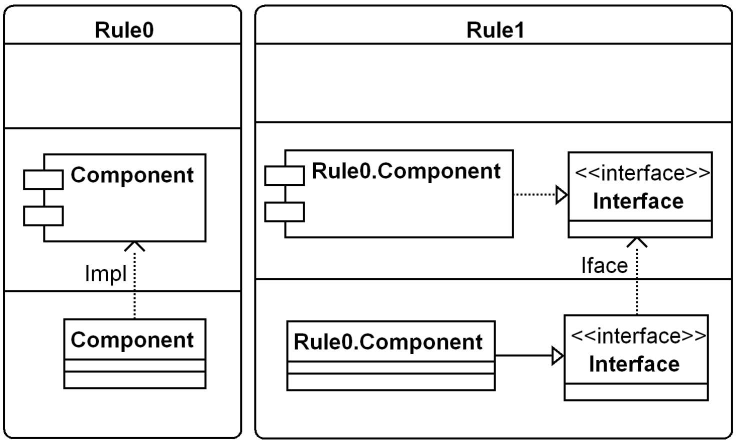


Figure 7.4: Chained transformation rules in UML-style

the meta model.

However, object diagrams in Kafka rules are more than just shortcuts for experienced developers. They are mandatory if either the PSM or the PIM – or both – have no associated graphical notation. Section 12.5 introduces a meta model for CORBA CCM. This meta model does not have a graphical notation yet. Using object diagrams a graphical notation is not required for the use of Kafka.

7.3 Mapping Graph

In subsection 6.3.2 the importance of the mapping graph for round-trip engineering has been highlighted. The mapping graph must be shaped in such a way that the generated IDs of two different vertices in the PSM are different.

$$ID(s_1) = ID(s_2) \;\Leftrightarrow\; s_1 = s_2$$

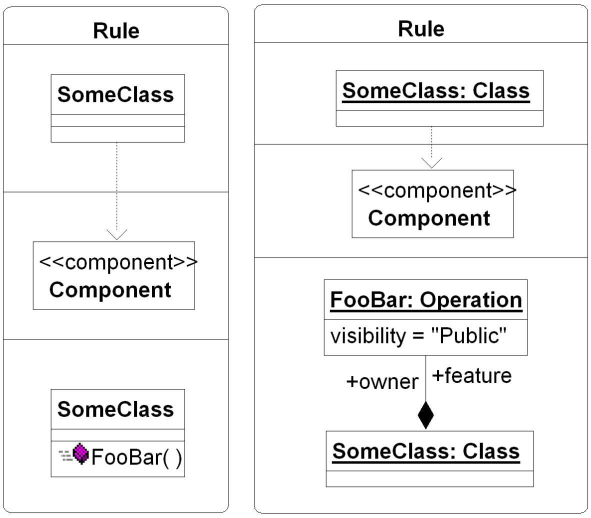


Figure 7.5: Two equivalent Kafka rules using different notation

$S = (V_S, E_S)$ = PSM
$s_1 \in V_S$ = vertex of the PSM
$s_2 \in V_S$ = vertex of the PSM
ID = as in definition 6.2

The ID should ideally describe why a vertex in the PSM has been created. The reason why a Kafka transformation created a PSM vertex consists of:

- the rule that has been applied while the vertex is created,
- the model element in the replace pattern that caused the vertex to be created,
- the PIM vertices that matched the PIM search pattern, and
- the PSM vertices that matched the PSM search pattern.

For every vertex v of the PIM that matched the search pattern, a new vertex r_v can be introduced in the mapping graph R. The vertex r_v is connected via one edge with v and via the other edge with the created PSM vertex s. The label of r_v is the name of the Kafka rule. The PSM vertices that matched the PSM search pattern cannot be included in the same way as the PIM vertices since the mapping graph must not connect two PSM vertices. These PSM vertices have been created by previous Kafka rules. Hence, they already have a unique ID. This ID can be encoded as a string to serve as a label for a new vertex r_v in the mapping graph. This vertex r_v is connected to the top-level vertex of the PIM, hence, the vertex that is the root of the PIM's spanning tree. In summary, for every vertex v in the PSM search pattern, a new vertex r_v is added to the mapping graph R. The vertex r_v is connected via one edge with the top-level PIM vertex and via the other edge with the created PSM vertex s. The label of r_v is $ID(v)$ followed by the rule name.

To distinguish PSM vertices created during the same application of the same rule, the model element in the replace pattern that caused the vertex to be created must be known and its identity must be encoded in the mapping graph. Each model element in a Kafka rule has a

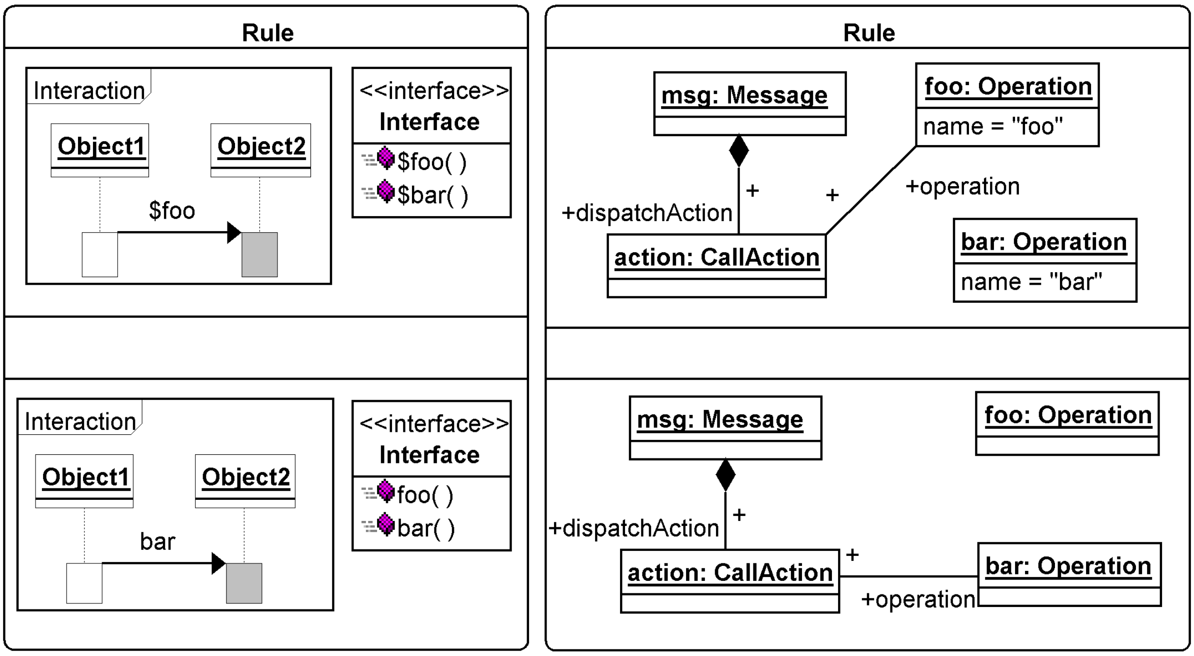


Figure 7.6: Two almost equivalent Kafka rules for the manipulation of sequence diagrams

unique numeric ID (not to be confused with the *ID* function of definition 6.2) that has been created by the UML tool (i.e. Kase) during construction of a rule. This ID is appended to the label of all vertices r of the mapping graph if r is connected to the created PSM vertex s.

Figures 7.7 and 7.8 illustrate the creation of the mapping graph. The PIM search pattern of the rule contains three vertices (interface, component, port) and the PSM search pattern one vertex (an interface). The rule creates one vertex (a class). Therefore, the mapping graph will have four vertices connected with this class. Three end up in the three PIM elements as shown in Figure 7.8. Each vertex of the mapping graph is displayed as a dashed line with an arrow. The label is visible, too. It contains the name of the rule (here: `Rule`) and the ID (here: `Component`) of the rule element that caused the creation of the PSM class. The vertex of the mapping graph that has been created because of the PSM search pattern's vertex is connected with the PIM top-level vertex – the PIM model itself. Its label has an additional prefix: the stringified ID of the PSM vertex that matched the search pattern.

This creation of the mapping graph yields unique IDs for every created PSM vertex. A shortcoming of this automatic approach is that two vertices in PSM models S_j and S_{j+1} may have different IDs although they have been created for the same reason. This can happen if two different rules do almost the same. For example, one rule maps a PIM component without provided ports to a PSM class. A second rule maps a PIM component with provided ports to a PSM class. If the PIM component in PIM I_j did not have provided ports, the first rule applied. If the new version I_{j+1} features a provided port, the second rule applies. This results in different IDs for the two generated classes in S_j and S_{j+1} although they should ideally have the same ID.

In such cases the rule developer must model the mapping graph manually. Both rules could

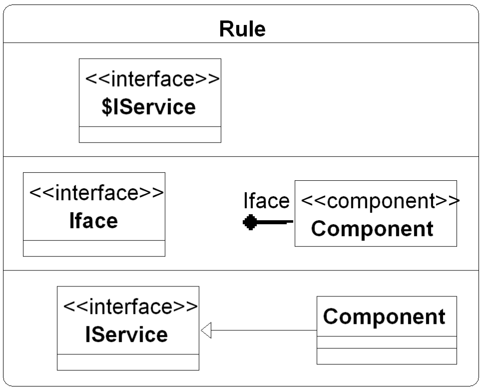


Figure 7.7: A Kafka transformation rule

create a vertex r labeled with some string such as `IMPL`. The vertex r is connected with the PIM component and the PSM class. In this case it does not matter whether the first or the second rule is applied. The ID of the generated class will be the same.

7.4 Embedded Python

The notation presented in section 7.1 and the Python-based transformation (see subsection 4.5.4) are complementary techniques. Many transformation problems can be easily addressed with graphical rules. Others are better tackled with a textual language. Kafka combines both approaches. The structure of a Kafka rule adheres to the principles of graph transformation. In addition, Python expressions can be embedded in rules. These embedded Python expressions serve three purposes.

- Python expressions can be used to customize the search pattern matching process. Hence, they act as application constraints in the sense of definition 4.2.
- They can apply further transformations to the PSM every time the search pattern matching succeeded.
- They can be used to compute attribute values – i.e. labels.

7.4.1 Application Constraints

The first kind of embedded code is a form of the application constraint *appl* mentioned in definitions 4.2 and 5.6. Such an expression returns a boolean value. If the return value is *true*, the rule is applied, otherwise not. The expression is always evaluated *after* an occurrence of the search patterns has been found and *before* the PSM search pattern is replaced by the PSM replace pattern.

All examples provided so far act on structural properties and typing of the model. In terms of definition 5.3 this means that the vertices V, the edges E, and their typing functions t_V and t_E are required during application of transformation rules. Embedded code can be used

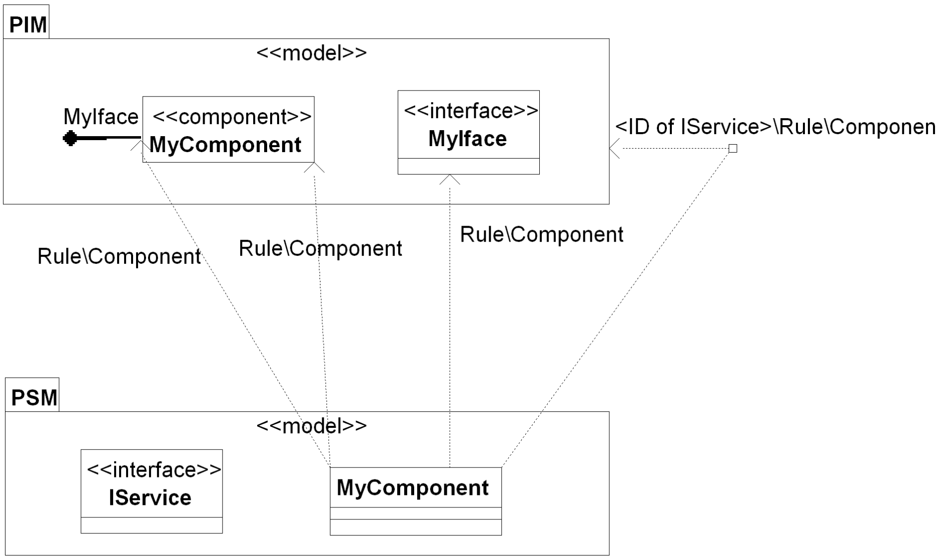


Figure 7.8: The result of one application of the rule shown in Figure 7.7

to fine tune the search pattern mapping by including the attribute values, i.e. the labels of the vertices.

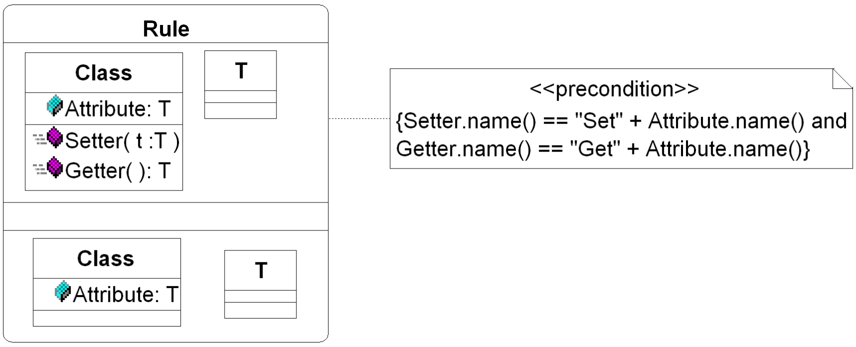

Rule
Class
Attribute: T
Setter(t :T)
Getter( ): T
T
<<precondition>>
{Setter.name() == "Set" + Attribute.name() and
Getter.name() == "Get" + Attribute.name()}
Class
Attribute: T
T

Figure 7.9: Example of an application constraint

The UML concept of a precondition is used to model application constraints. The embedded Python code can refer to elements in the search patterns by name. Figure 7.9 provides an example. The transformation rule searches for getter and setter functions of an attribute and removes them. For example, the expression `Getter.name()` refers to the name of the operation that matched the model element `Getter` of the PSM search pattern.

Application constraints cannot access elements in the replace pattern, because they do either not exist when the patterns have been matched (because they are created during the application of the rule) or they have a correspondence in the target pattern (provided by the mapping map_{LR}). In the first case it would be an error to attach an application constraint to an element that is created only if the rule was applied. In the second case the element could as well be attached to the corresponding element in the search pattern.

7.4.2 Fix Up Expressions

Graphical transformation rules allow to change the structure of a model, hence, the edges E, vertices V, and the typing functions t_V and t_E. However, the vertex labels need special treatment, because vertices and their labels are simply copied from the replace pattern into the PSM. The label of a vertex x includes a value for each meta attribute of the meta class $t_v(x)$, i.e. for each meta attribute $a_{\mathcal{M}}(t_v(x))$. Therefore, these meta attribute values are static from the viewpoint of a graph transformation engine. This means that they do not depend on the meta attribute values of the elements that matched the search pattern. Fix up expressions provide means for dynamic meta attribute values. However, if a fix up expression creates a new model element – a procedure that is strongly discouraged – it must build the reference graph manually. Otherwise this may break the forward merging algorithm.

In Kase, fix up expressions are sequences of Python statements. These statements are executed after the replace pattern has been glued in and connected with the model. The Python statements can access the elements of all three patterns.

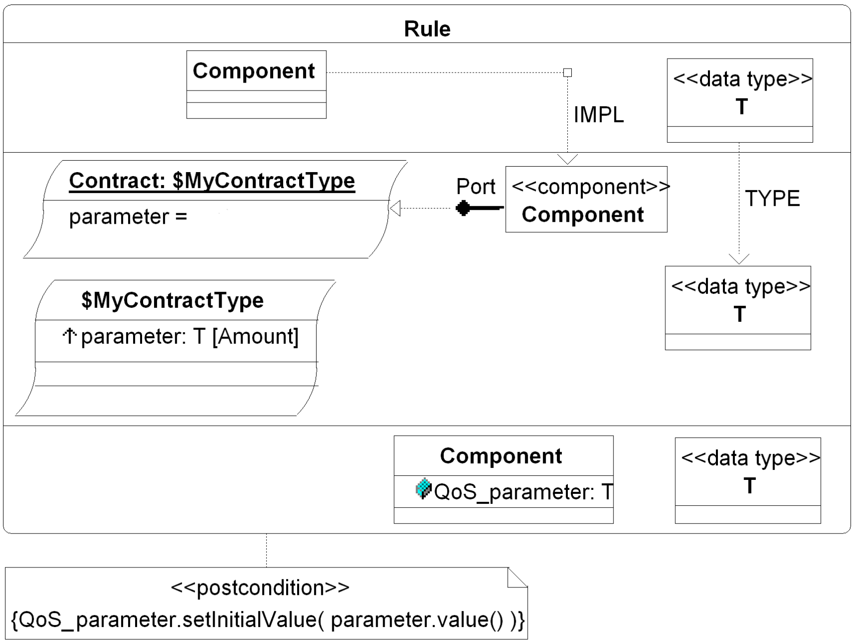


Figure 7.10: Example of a fix up expression

The example in Figure 7.10 creates a UML model element named `QoS_parameter` in the PSM if the component realizes the contract type `MyContractType`. The fix up expression sets the value for the meta attribute `InitialValue` of the created model element `QoS_parameter` to the value of the PIM QoS `parameter` specified in the contract.

7.4.3 Attribute Expressions

All text attributes of elements in the replace pattern of a rule are fed through the Python interpreter. This interpreter searches for Python expressions located between two `@` characters. The expression is evaluated and replaced in the text by the result of the evaluation. For example, the string

```
The number @21 * 2@ is the product of 21 and 2
```

will be replaced by

```
The number 42 is the product of 21 and 2.
```

Such Python expressions can access the PIM and PSM model elements that matched the current rule and the elements created by the PSM replace rule. Figure 7.11 provides an example. It adds two operations – a constructor and a destructor – to a class. Constructors and destructors must have the same name as their class. Therefore, the expression `@Class.name()@` has been inserted in the attribute value of the two operations. This expression is replaced by the name of the class whenever the transformation rule is applied.

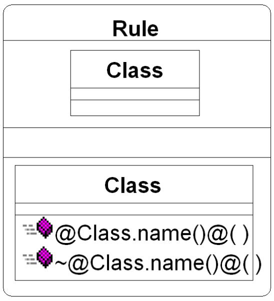


Figure 7.11: Example of attribute expressions

From a theoretical point of view attribute expressions are superfluous. They can always be replaced by fix up expressions. They are just provided for convenience.

7.5 Orchestration

A Kafka transformation contains several Kafka transformation rules (see definition 5.6). These rules are orchestrated. The orchestration determines in which order the rules are to be applied. In contrast, graph rewriting theory does not provide an orchestration. There it is the task of the rewriting engine to determine the next applicable rule. As discussed in subsection 7.6.1 this may lead to ambiguousness. Therefore, Kafka introduces the concept of orchestration.

The orchestration is depicted as a UML state chart. The major difference is that the states are transformation rules. Figure 7.12 gives an example.

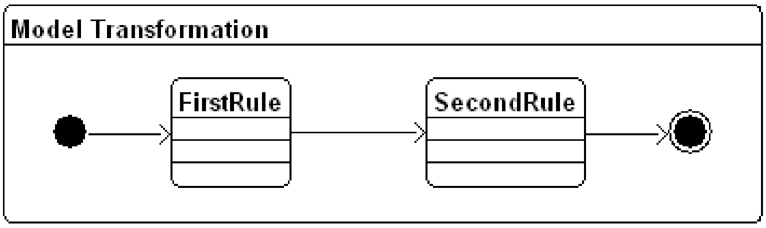


Figure 7.12: Simple orchestration without alternative paths

Each transformation starts at the initial state vertex and traverses the state machine until the final state vertex is reached. When a new state – e.g. rule – is entered, this rule is executed. Then the outgoing edges are checked. An edge may have a Python expression that returns a boolean. If it returns *true*, this edge is traversed. Otherwise the unlabeled edge is traversed. To guarantee the termination of the algorithm 7.1, the orchestration must not contain loops. Furthermore, every vertex – except the final state vertex – must have exactly one outgoing edge without a Python expression. This is the default edge. Typically, the orchestration just puts the set of rules in an absolute order as shown in Figure 7.12 The rules are applied one after the other. Sometimes it is required to introduce an alternative path through the set of rules. For example, the same Kafka transformation could be used for internal projects of a company and for customer projects. Depending on the type of project, some differences may be necessary. Instead of maintaining two transformations, the same transformation can detect the kind of project and choose different transformation rules in some places. This is illustrated by Figure 7.13.

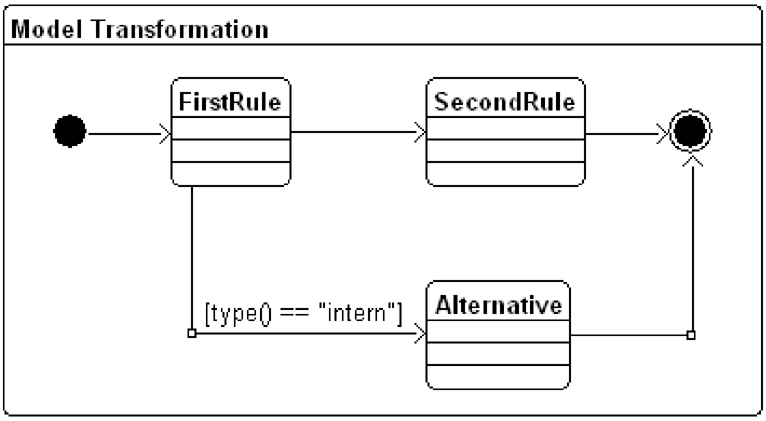


Figure 7.13: Orchestration with one alternative path

7.6 Transformation Algorithm

A complete Kafka transformation consists of a set of rules and their orchestration. The following definition provides a precise definition.

Definition 7.1. A *Kafka transformation* T is a directed acyclic graph $T = (V, E)$.

The vertices V are MDA model transformation rules according to definition 5.6.

The edges E represent the orchestration.

V contains two dedicated vertices: an *initial state vertex* and a *final state vertex* f.

The edges $E \subseteq V \times V$ can be labeled. A label is a boolean Python expression that acts on an input model G.

For each edge e the function $\ell_e : G \mapsto \mathbb{B}$ denotes whether the Python expression returns true when applied to graph G. An edge may be unlabeled. In this case ℓ_e returns always *true*.

For every $v \in V \setminus \{f\}$ the property

$$\mid \{ \ e \ \mid \ e = (v, v_i), \ v_i \in V, \ e \text{ is not labeled } \} \ \mid = 1$$

must hold. Hence, each vertex except f has exactly one unlabeled outgoing edge. Furthermore, the Python expressions must be mutual exclusive for every input model G.

$$\forall \, G : \ \mid \{ \ e \ \mid \ e = (v, v_i), \ v_i \in V, \ e \text{ is labeled}, \ \ell_e(G) = \mathit{true} \ \} \ \mid \leq 1$$

The Kafka transformation algorithm transforms an input model G into an output model H by applying the Kafka transformation T.

$$G \Rightarrow_T H$$

G and H are models in the sense of definition 5.3

Algorithm 7.1. The Kafka algorithm transforms an input model G by applying a Kafka transformation $T = (V, E)$.

T is by definition a directed acyclic graph and its vertices V are MDA model transformation rules according to definition 5.6.

- r := initial state vertex $\in V$
- determine $e = (r, r_i) \in E$ for which $\ell_e(G) = \mathit{true}$
- if $e = \mathit{none}$ then find the unlabeled edge $e = (r, r_i) \in E$

```
• $r := r_i$ with $e = (r, r_i)$
• while $r \neq f$ (i.e. the final state is not reached)
    – $M := \{ J \mid J \subseteq G \wedge J$ is a match of $L_r$ in $G \}$
    – remove all $J \in M$ for which $appl_r$ is not satisfied (see subsection 7.4.1)
    – foreach $J \in M$
        ∗ if $J$ is still a match of $L_r$ in $G$ and $appl_r$ is satisfied
            · transform $J \subseteq G$ with rule $r$, i.e. $G \Rightarrow_r G_{tmp}$
            · apply fix up expressions to $G_{tmp}$ (see subsection 7.4.2)
            · $G := G_{tmp}$
```

G = the model that is to be transformed
$T = (V, E)$ = a Kafka transformation
$r \in V$ = $(\mathcal{G}, \mathcal{S}, \mathcal{R}, L, R, K, map_{KR}, map_{LR}, appl)$
$f \in V$ = final state vertex
$appl_r$ = application constraint *appl* of the current rule r
L_r = left-hand side L of the current rule r
M = a list of sub-graph matches in G

The algorithm traverses the vertices of V. Each vertex is a model transformation rule that is applied when the vertex is entered. The application of the rules is realized in two steps. First, all possible matches M are determined. Then a loop iterates over the matches in M and modifies G step by step. Alternatively, one could imagine that the algorithm finds one match, modifies G, searches the next match, and so on. As shown in the next section the chosen approach guarantees that a Kafka transformation always terminates.

7.6.1 Termination and Unambiguity

A complete graph transformation contains a set of transformation rules. In general, graph transformations do not necessarily describe the order in which these rules are applied. A rule $r = (L, R, K, map_{LR}, map_{KR}, appl)$ is applied if the left-hand side L could be found and if the application constraint *appl* is satisfied. The inherent problem of this approach is that ambiguities become more probable. Furthermore, a graph transformation does not necessarily terminate. This is shown by the following two examples. A set of rules

$$P = \{r_1, r_2\}$$

is applied to a start graph G_0. The following derivations are possible.

$$G_0 \Rightarrow_{r_1} G_1 \Rightarrow_{r_2} G_2$$

$$G_0 \Rightarrow_{r_2} G_3 \Rightarrow_{r_1} G_4$$

G_1 may differ from G_3. However, G_2 and G_4 can be equal nevertheless. If G_2 and G_4 are final derivations and $G_2 \neq G_4$ then the transformation is ambiguous. A graph transformation system that does *not* allow such ambiguous derivations is called *confluent*.

A non-terminating transformation can be constructed as follows. Let r_1 replace a vertex labeled `A` with a vertex labeled `B`. Let r_2 replace a vertex labeled `B` with a vertex labeled `A`. If G is a graph with one vertex labeled `A` then

$$G_0 \Rightarrow_{r_1} G_1 \Rightarrow_{r_2} G_2 \Rightarrow_{r_1} \Rightarrow_{r_2} G_3 \Rightarrow_{r_1} \cdots$$

is an infinite derivation.

Research in the field of graph transformation theory has investigated the problem of ambiguity and infinite derivations. Termination in general is undecidable [Plu98]. However, as shown later in this section the termination of Kafka rules is decidable. Some approaches exist to detect ambiguity of graph transformation systems. They are based on *critical pairs* [Plu93].

Definition 7.2. Let $r_i = (L, R, K, map_{KR}, map_{LR}, appl)$ be a set of transformation rules, for $i = 1, 2$. Let g_i be a set of morphisms. A graph G is constructed such that $G := g_1 L_1 \cup g_2 L_2$. A pair of direct derivations $G_1 \; {}_{g_1,r_1}\!\Leftarrow G \Rightarrow_{g_2,r_2}$ is called a *critical pair* if $g_1 L_1 \cap g_2 L_2 \neq g_1 K_1 \cap g_2 K_2$ and if either $r_1 \neq r_2$ or $g_1 \neq g_2$.

Hence, a derivation pair is critical if the left-hand sides of r_1 and r_2 overlap and if one rule removes a vertex or edge from this overlap. If $g_1 L_1 \cap g_2 L_2 = g_1 K_1 \cap g_2 K_2$, the overlap is not altered and the pair is not critical. In Kafka, the detection of critical pairs is reduced to the special case $r_1 = r_2$ since the orchestration determines the order in which rules are to be applied. However, it can happen that $g_1 \neq g_2$, i.e. the left-hand side of one rule has different matches in G. In this case, a critical pair is possible. This is proven by the following example. The graph G_0 contains two vertices labeled with `A` and one labeled with `B`. The graph has only one edge. G_0 connects one `A` vertex with the `B` vertex. Thus, the two `A` vertices can be distinguished. The rule r searches two `A` vertices and removes the first one. Hence, the algorithm has two possibilities for the first application of r. After the first application $G_0 \Rightarrow_r G_1$ the graph G_1 contains only one `A` vertex. This implies that G_1 is irreducible – r cannot be applied a second time. However, which `A` vertex has been removed is a random outcome. Therefore, a critical pair exists and the transformation system is not confluent.

[Plu93] shows how to detect critical pairs to decide whether a transformation system is confluent. Furthermore, [Plu93] proves that a finite and terminating (hyper-)graph transformation system without critical pairs is strongly confluent. However, a confluent, finite, and terminating transformation system can contain critical pairs. Thus, enforcing the elimination of critical pairs can guarantee that the transformation system is confluent, but this constraint is too restrictive. It may rule out confluent transformation systems. [Plu93] even showed that it is undecidable in general whether a finite, terminating (hyper-)graph transformation system is confluent. A further source of complications is Kafka's use of Python. The Python code can insert, remove, or modify the graph. This has proven to be a useful feature in practice. The drawback is that it is impossible to decide a priori what a Kafka rule will do since that would require to understand all implications of the embedded Python code. The problem is illustrated by the following example. Let G_0 be defined as above. The rule r searches one `A` vertex and labels it with k. The value k is increased by the Python code by one after each application. Hence, k is a side effect introduced by the Python code. In the derived graph one vertex will be labeled with `1` and one with `2`. Which `A` vertex is relabeled to `1` is accidentally. Therefore, the detection of critical pairs would have to ignore the embedded Python code.

In summary, Kafka does not provide means to detect the ambiguity of a rule. Especially the embedded Python code renders an automatic detection of ambiguousness impossible. This is a tradeoff between ease of use on one hand and verification capabilities on the other hand. In the context of this thesis the transformation is a tool. Therefore, the ease of use got precedence over verification capabilities. If a developer implements a transformation directly in Python, he can face the same problem. The simple loop

```
for x in doc.getClass() :
    transformClass(x)
```

can already be ambiguous if `transformClass` has unexpected side effects. However, the problem of termination can be solved for Kafka transformations.

Proposition 7.1. The Kafka transformation algorithm 7.1 terminates for every Kafka transformation T and for every input model G.

Proof. $T = (V, E)$ is directed and acyclic by definition.
$\Rightarrow$ Each directed path in T visits a finite number of vertices.
$\Rightarrow$ The outer loop of the algorithm exits.

G^* is the current model – e.g. a graph – and a vertex $r \in V$ (i.e. a Kafka rule) is entered. Then r is applied as often as possible.

$$G^* \Rightarrow_r G_1 \Rightarrow_r G_2 \Rightarrow_r G_3 \Rightarrow_r \cdots$$

The algorithm starts by determining the set M of all possible applications of r to G^*. G^* has a finite number of vertices $\Rightarrow$ M is a finite set.
$\Rightarrow$ the sequence of derivations terminates, because r is applied at most once for each $J \in M$.
$\Rightarrow$ the inner loop of the algorithm exits.
$\Rightarrow$ the algorithm terminates. $\square$

The proof relies on the fact that the algorithm first determines the set of matches M and executes the derivations afterwards for every $J \in M$. If the algorithm would instead find one match of L_r in G, execute $G \Rightarrow_r G_1$, search the next match, execute $G_1 \Rightarrow_r G_2$ and so on, then the algorithm does not necessarily terminate. Each application of the rule r could introduce a new sub-graph that is a match of L_r in G. Hence, the rule r would be applied infinite times.

7.6.2 Time and Space Complexity

The termination property of the algorithm leads to the following proposition.

Proposition 7.2. The size of the resulting graph H of a Kafka transformation T

$$G \Rightarrow_T H$$

is polynomial in the size of G if the size of T is treated as a constant. Furthermore, the Python fix up expressions of T are assumed not to create new vertices and to execute in constant time.

Proof. $T = (V_T, E_T)$ has a finite number of rules. $t :=\mid V_T \mid$ can be treated as a constant.

Each rule $r \in V_T$ has a left-hand side $L = (V_L, E_L)$. The maximum of all $\mid V_L \mid$ is treated as a constant called l in the following.

$G = (V_G, E_G)$ is the input graph and $n :=\mid V_G \mid$ is treated as a variable.

If a rule r is applied to the graph G, a constant number of maximum c additional vertices and edges is added. It remains to be proven that r is applied $O(n^l)$ times. Then the added vertices and edges are in $O(t \cdot c \cdot n^l)$ with constants c, l, and t.

In the worst case each vertex in V_L matches each vertex in V_G. This leads to

$$\frac{n!}{(n-l)!} = n \cdot (n-1) \cdots (n-l+1)$$

possible matches. The correctness of the proposition follows from the estimate

$$\frac{n!}{(n-l)!} \leq \prod_{i=1}^{l} n \in O(n^l)$$

and the fact that l is a constant. □

The worst case estimate in the above proposition is weak, because the constant l can become very large. Although $l \ll n$ is usually the case. The most sensible conclusion of the estimate is that a malignant Kafka transformation can easily exceed the available memory. The same approximation can be used to estimate the time complexity of the algorithm.

Proposition 7.3. The time complexity of algorithm 7.1 is polynomial in the size of input model G if the size of transformation T is treated as a constant.

Proof. The algorithm iterates over all rules r and for each rule it searches matches of the left-hand side L of the rule in the input model G. Finding the matches is the expensive part. Adding, modifying, or removing edges and vertices from G requires constant time for each application of a rule. From proposition 7.2 follows that the maximum number of matches throughout the entire transformations is in $O(t \cdot n^l)$ where l and t are constants. If the algorithm simply tries every possible match, the application of each rule is polynomial in n. □

Once again, this worst case estimate applies only to malignant transformations. The average is lower because the initial assumption that every vertex of a rule's left-hand side L matches each vertex in the input model G is usually wrong. For example, if L searches for a class and an operation, it has two vertices. The class vertex will only match those vertices in G that are classes, too. The same applies to the operation vertex. Thus, the initial assumption is very pessimistic. In practice the speed of transformations did not impose a problem. However, it was observable that the size l of the left-hand side L of a rule has significant impact on the runtime. This effect is in line with the result of proposition 7.3.

Chapter 8

Tool Chain

The MDA requires a specialized tool chain [WUG03d, WUG03c], including tools such as model editors, model transformers, model versioning tools, specialized debuggers, etc. If developers spend too much time fighting with inappropriate tools, the MDA will not speedup software development. Instead, the MDA would just introduce an additional overhead. Therefore, a specialized tool chain for the presented methodology has been developed. Figure 8.1 shows the different components of the tool chain and how they work on the models and source code.

The tool chain starts with an editor that allows developers to create the PIM. Then the model transformation follows. The transformation yields (internally) a QoS-agnostic PSM. This model is never visible to the user. It is just an intermediate model. The QoS aspects have to be woven with this intermediate model. The outcome is the QoS-aware PSM. Once again, a model editor is required to extend the PSM (if needed). If the PIM is an executable model, this step may be no longer required. However, executable models are still subject of intensive research. In the meantime, the generated PSM can be treated as kind of a design and implementation skeleton that has to be extended with respect to design and source code. Then, the PSM is input to a code generator. The code generator creates source code and perhaps deployment descriptors based on the PSM's deployment diagrams. The generated source code is loaded by an IDE. The IDE controls editing of the source code, compilation, and physical deployment.

The PSM QoS aspects must be created by experienced QoS developers. Hence, another editor is required for this task. In Kafka, these QoS aspects are represented as transformation rules. A Kafka transformation rule is itself a model. Thus, editing QoS aspects requires another model editor.

8.1 Model Editing Tool

From a user's point of view the most important tools are the model editing tools. A model editor is required for the PIM and for the PSM. If the PIM is standard UML extended with some profiles, a standard UML tool can serve for both kinds of models. However, if the PIM is built on a meta model different from UML or on a UML meta model extension, special tool support is required. Some commercial tools such as ArcStyler [Int03] use different editors for the PIM and the PSM. The drawback of this approach is that users have to learn two editors.

The tool chain developed throughout this thesis solves this problem by offering one model editing tool (called Kase) that is capable of editing standard UML models as well as models based on other meta models. The benefit is an easier to learn tool chain. Furthermore, the model transformation itself is integrated in the model editor. Developers do not need to switch from one tool to the other. The entire MDA process is hidden behind a consistent graphical user interface. All activities related to source code editing, compilation, and physical deployment are left to industry standard IDEs (Integrated Development Environment).

8.1.1 Extensibility

Kase has built-in support for UML 1.4. Furthermore, it supports some extensions introduced by UML 2.0, such as the new component model and HMSCs. Kase is extensible in various ways. It supports UML profiles. A profile is a collection of stereotypes and tagged values. They can be used to specialize concepts of the UML. For example, a CORBA CCM profile can specialize the concept of an interface by stereotyping it as an IDL interface. The drawback of profiles is that they cannot really add new concepts to the UML. However, some developers are tempted to abuse this extension mechanism. Instead of refining the semantics of a UML construct, they redefine it, which is not the original extension of stereotypes.

Another drawback of UML profiles is the ugliness of the resulting diagrams. Text sequences of the type $\ll$ StereoTypeName $\gg$ are scattered across the entire diagram. To overcome this problem, Kase supports notation plugins. Such plugins are hooked into Kase's diagram engine. If a stereotyped element is to be displayed or edited, the plugin can offer a special notation or editing support. This way standard profiles can be used and at the same time the notation can be significantly improved.

Especially for PIMs meta model extensions may be required. Kase has no built-in meta model. Instead, the meta model is read at runtime from an XML file. Kase's core is of course not able to display instances of the new meta classes in diagrams. Plugins solve this problem. They add notation and editing support for meta classes which are not part of standard UML.

Kase can store the PIM and the PSM in one physical model. To achieve this, both meta models are merged into one combined meta model.

8.1.2 Editing Kafka Rules

The merged meta model $\mathcal{H}$ (see definition 5.5) is especially important for the editing of Kafka transformation rules. Each rule uses elements of the PIM *and* the PSM. Furthermore, the execution of a model transformation builds the mapping graph between both models. Due to the combined meta model this is quite easy to realize. If PIM editing and PSM editing are realized by two non-combined meta models, editing of Kafka transformation rules would not be possible.

Kase does not only allow the editing of PIMs and PSMs. It is used for editing Kafka transformation rules, too. Kafka has its own meta model. This meta model contains concepts such as a rule, search pattern, replace patterns and references between PIM and PSM meta classes. Therefore, Kase operates on a combined meta model that merges PIM, PSM, and the Kafka meta model into one meta model.

A Kafka transformation (i.e. a set of rules and their orchestration) can be compiled into an executable Python script. A tool called SAMSA (named after the protagonist in Franz Kafka's book Metamorphosis [Kaf16]) handles this compilation step. To execute the transformation, the Python script is executed in Kase via a built-in Python interpreter

8.2 Python Scripting

Kase utilizes Python as a general purpose extension language. However, the primary task of the interpreter is to execute the transformers, which SAMSA has compiled from Kafka transformations. Other areas of application are source code and document generation, model checking, and model refactoring [FBB$^+$99, SPTJ01].

The Python interpreter is embedded in Kase. This design has some advantages over XMI based tools. First of all, the diagrams adapt automatically to changes in the model. If the model is first exported as XMI, then transformed and re-imported as XMI, the diagram information is usually lost. This might be resolved with UML2 because the diagramming information itself will be based on a meta model [OMG03h] and can be exported as XMI, too. However, it would still be the task of the transformation to keep the diagramming information in sync with the model changes. In Kase this is achieved without cooperation of the transformers. The diagram objects listen to events generated by the model. As the model changes the diagrams adapt step by step. This simplifies the development of transformers since the diagrams are automatically kept in sync without increasing the complexity of the transformer. Furthermore, it is possible to suspend transformations. This is important for debugging since the developer of a model transformation can look at intermediate results. However, this feature has not yet been implemented although it is technically possible.

Kase has a built-in undo/redo mechanism that listens to model and diagram changes. These changes can be reverted and re-played. This mechanism works independently of the kind of changes applied to the model or diagrams. Even complete model transformations or model refactoring can be handled by the undo/redo mechanism.

Python is an interpreted language. Hence, the idea of adding an interactive Python shell to Kase was a logical step. The advantage of an interactive shell is that developers can easily try out some ideas. The diagrams adapt automatically after each command is completed. Hence, the developer can follow step by step the results of each command. The interactive shell provides advanced concepts such as command completion. Python's reflection capabilities are used to inspect the API and complete commands as they are typed in. This is especially useful for those developers who do not know the meta model - and hence the Python API - by heart, since the Python API generated for UML 1.4 contains 1461 methods.

The interpreter is used for model checking, too. A special Python script checks the UML well-formedness rules. If the model is not well-formed, a dialog is displayed that highlights the erroneous model elements and provides advice. New meta models require checking new well-formedness rules. It is very convenient that the checking of these rules is not hard coded in Kase.

Furthermore, Python expressions can be embedded in the model as a special kind of constraints. Standard UML constraints have been designed to check instances of models, i.e. running systems. As a typical example, a constraint might limit the balance of a bank account to a value ≥ 0. To evaluate such a constraint an instance of the bank account class is required. Evaluation of constraints happens at runtime. In contrast, the special kind of constraint checks the model and not instances of the model. This constraint is kind of a well-formedness rule embedded in the model. These constraints are usually created throughout a model transformation. This has the following reason: The developers can change and enhance the generated PSM. For example, they can deploy components in a deployment diagram. These changes cannot be controlled by the model transformer. Nevertheless, the transformer can create special constraints that ensure a correct deployment. For example, a constraint may enforce that some component is deployed on three redundant machines in order to achieve a high availability. This idea is exploited in section 12.1.

8.3 Model Access API

The embedded Python interpreter can access the model. This requires a special API. Two different approaches for designing such an API are possible. Option one is based on reflection. This approach provides full read access to the meta model. To access vertices, edges or labels of the model the respective meta model constructs are required. The following example illustrates this.

The fictive transformation engine – built on such a reflective API – has, for example, a reference to an operation $v \in V$. The operating belongs to a model $M = (V, E)$ that is based on the meta model $\mathcal{M} = (V_{\mathcal{M}}, E_{\mathcal{M}})$. To traverse the association between the operation and its parameters (see Figure 5.1) the engine must find out the meta class of v. This will yield `Operation`. Then it must find the required meta association between the meta classes `BehavioralFeature` - which is a super class of `Operation` - and `Parameter`. Now the engine can query for all edges in M that have the required type - where type means meta association. The advantage of this approach is that the same API works for every meta model. The drawback is that the usage of the API is very difficult as the above example demonstrated.

The second approach does not use reflection. Knowledge about the meta model is encoded in the API. This simplifies the usage of the API. The example discussed above is repeated here, this time using the non-reflective API. Again, the engine has a reference to an operation $v \in V$. The API provides a function that allows traversing from an `Operation` to its `Parameters`. The difference here is that only one API call is sufficient. In contrast, the reflective API required three calls.

A drawback of the non-reflective API is that a new API is required for every new meta model. Furthermore, meta programming is not possible. For example, there is no way of determining the spanning tree G_{span} of M independently of $\mathcal{M}$. However, if $\mathcal{M}$ is known a priori then it is possible.

The current implementation of Kase uses a non-reflective API. The reason for this decision was the ease of use. The reflective API is very hard to use and its only major advantage is its meta model independence. To cope with a special tool has been developed that automatically generates a non-reflective Python API for a given meta model. The most important information required for the generation of the API are:

- the meta classes $V_{\mathcal{M}}$,

- the meta associations $E_{\mathcal{M}}$, and
- the meta attribute mapping $a_{\mathcal{M}}(v)$.

This information is extracted from the meta model and implanted into the generated Python API. The Python API allows for creation and deletion of model elements. Just as in the XML Document Object Model [W3C98] (DOM) a `Document` class is used for this purpose. It has a factory function for every meta class in $V_{\mathcal{M}}$. Furthermore, every meta class $\in V_{\mathcal{M}}$ is represented as a Python class. If two meta classes participate in a generalization relationship, this applies to the respective Python classes, too. This approach causes some problems for languages such as Java and C# since they do not have multiple inheritance. A discussion of this problem can be found in [Bre02].

The Python pendant of a meta class v is equipped with getter and setter functions for all meta attributes of v. This way $a_{\mathcal{M}}(v)$ is encoded in the API. Furthermore, functions for traversing, checking, appending and removing instances of meta associations are added. Hence, $E_{\mathcal{M}}$ is encoded in the API. An excerpt of the generated API illustrates this.

```
class GeneralizableElement ( ModelElement ) :
    def isRoot( self ) :
...
    def setIsRoot( self, value ) :
...

class Classifier ( Namespace, GeneralizableElement ) :
    def removeFeature( self ) :
        ...
    def removeFeature( self, element ) :
        ...
    def hasFeature( self ) :
        ...
    def hasFeature( self, element ) :
        ...
    def feature( self ) :
        ...
    def appendFeature( self, element ) :
        ...
```

The excerpt shows the getter and setter functions for the *isRoot* meta attribute. The mapping of meta associations is shown by the $feature$ meta association of the $Classifier$ meta class.

The same design principles are used for a C++ model access library. This library is used inside the modeling tool Kase because it is very fast. Just like the Python API it is generated from the meta model following the same principles as the generation of the Python API. A code comparison will show that statically typed languages of the C-family such as C++ [Str00], C# [ECM01a], or Java complicate the API.

The following example shows how to create a list of all classes in the model. In Python this is a one line job.

```
[ x.toClass() for x in doc.allElements() if x.isClass() ]
```

In C-style languages the same construct would require many lines more. For example the C++ pendant looks like this.

```
list<UMLClass> result;
list<UMLModelElement> lst = doc.allElements();
list<UMLModelElement>::iterator it = lst.begin();
for( ; it != lst.end(); ++it )
    if ( (*it).isClass() )
        result.append( (*it).toClass() );
```

The Python statement and the C++ code block are built on the same API. Hence, the difference in complexity is a direct result of the language differences. Python features the concept of named parameters. When a new instance of a meta class is to be created then all attributes are set to their default value. However, the factory function accepts a list of named parameters to easily set the attribute values.

```
c = doc.createClass( name=''MyClass'', isAbstract=1 )
```

The same job implemented in C++, which does not have this feature, covers already three lines.

```
UMLClass c = doc.createClass();
c.setName( ``MyClass'' );
c.setIsAbstract( 1 );
```

An advantage of a Python based model transformation process in contrast to XSLT or OCL is that Python is a general purpose programming language. It can easily solve common tasks such as file IO, XML parsing, HTTP requests etc. Therefore, model transformations could for example contact a model repository to retrieve design or code fragments. Another application is the creation of a transformation log that lists the transformation steps, warnings, and error messages.

Throughout this thesis, many lines of source code have been written for traversing and modifying a model. It turned out that the use of Python reduced the code size and complexity tremendously. However, a drawback is that type errors occur during runtime only. Nevertheless, C-style languages including C++, C# and Java should only be used if performance is a very important requirement.

8.4 Code Generation

Kase itself has no special support for any target programming language. If a target programming language requires special modeling constructs, they can be added via profiles. For example, Kase ships with a C# profile. Some frameworks such as Microsoft's .NET may require additional profiles. Its language-agnostic design implies that the core of Kase is not able to generate source code.

Generation of source code is handled by special Python scripts. These scripts traverse the model and generate source code for them. The real challenge is not to generate the source code. Instead, the synchronization between a changed model and manually changed source code is the real challenge. Usually this requires a parser for the target programming language. Currently, Kase supports C# and Java as target programming languages.

Other Python scripts implement interoperability with IDEs. Currently, Kase ships with a script for Microsoft's VisualStudio.NET. This script can parse UML 1.4 style component and deployment diagrams to determine which classes and other artifacts have to be bundled to libraries or executables. This information is used to generate a complete VS.NET solution. Hence, the output of source code generation is not just a bunch of source code files. The code generation step can generate a complete solution including makefiles, source code, and deployment descriptors.

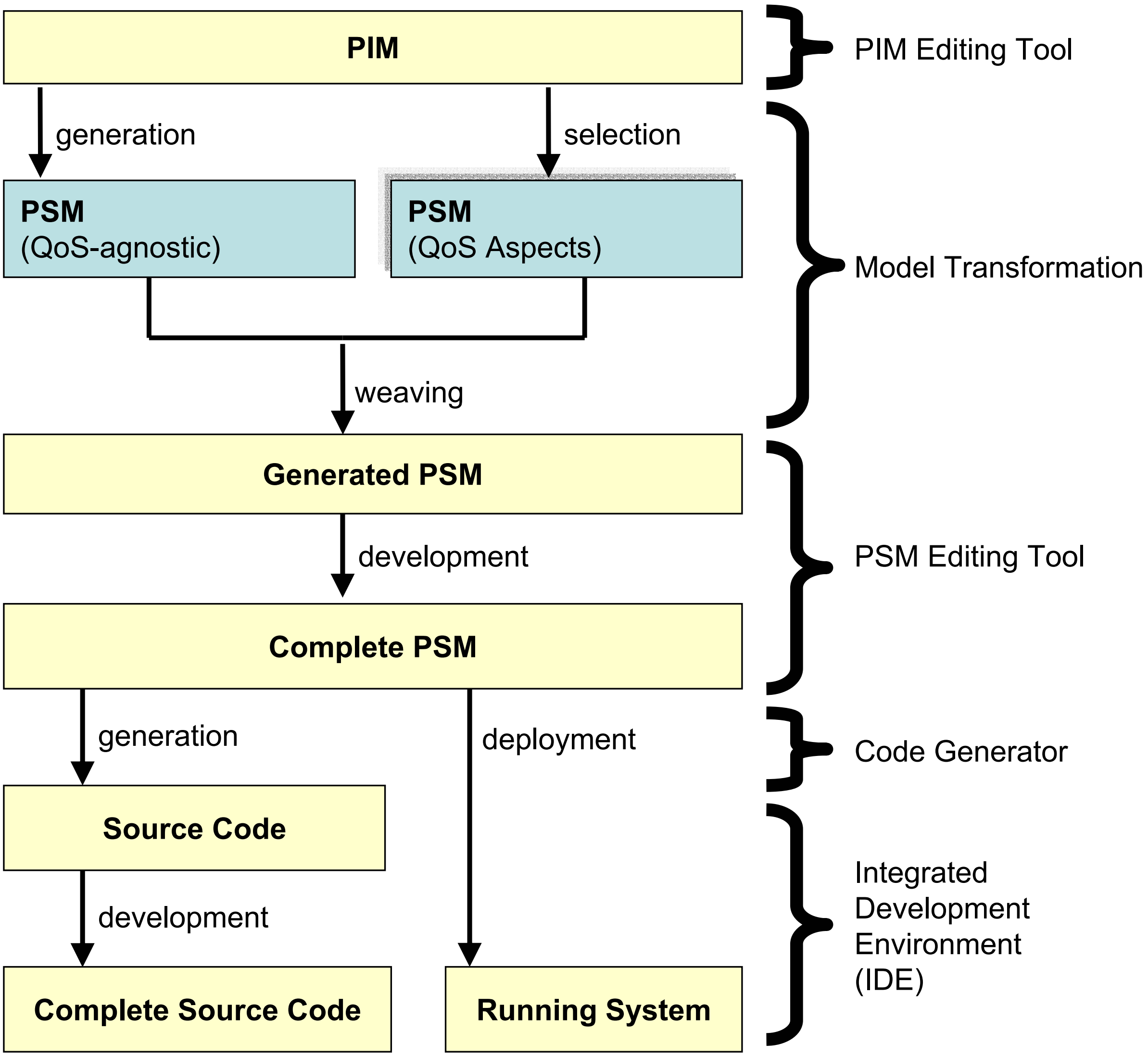


Figure 8.1: Tool chain for MDA

Part III

QoS Models

Chapter 9

Introduction

Part II focuses on the tooling problems of the MDA and presents a tool chain and the associated theoretical foundations. Part III discusses the modeling of QoS-enabled distributed applications and how the tool chain can be utilized to transform PIM QoS contracts into their PSM counterparts. The PIM is a model of the application described in terms of the application domain. The purpose of the PIM is to provide a high-level platform independent view on the application with a special focus on its QoS properties. The PIM shows *which* QoS properties components offer and require. Therefore, the PIM is highly declarative. In contrast, the PSM is a model of the system realization. It shows *how* the application realizes the QoS properties on top of a specific target platform.

The platform independence of the PIM is especially useful for the modeling of heterogeneous distributed applications. Usually, large scale enterprise applications operate in a heterogeneous environment. Such applications are built on a set of distribution platforms, operating systems, and programming languages. For example, web applications utilize three communication protocols: HTTP between web browser and web server, CORBA/EJB/.NET Remoting between web server and business logic, and a vendor specific protocol between business logic and database engine. By definition, the PIM abstracts from this set of technologies. Hence, the PIM is not obfuscated by details and scurrilities of the target platforms.

A large distributed application is composed of several services. The QoS properties provided by one service may be limited by the QoS properties it perceives from other services. For example, the response time of a web application is limited by the response time of the back-end. Such dependencies are important in order to understand how the application performs. By making such dependencies explicit in the model, developers can more easily identify possible bottlenecks. Furthermore, they can understand better how the application will adapt if resources such as network bandwidth or processing power vary. Therefore, the PIM modeling language presented in chapter 11 will support the modeling of QoS properties and their interdependencies.

In an ideal case an application can guarantee certain QoS-properties. This involves the allocation of resources such as bandwidth, processing power, and memory. Such applications are called *pro-active*. However, guarantees are often hard to achieve. Most operating systems do not provide any guarantee for processing power, timeliness, or memory. Furthermore, today's dominant network technology does not support bandwidth reservations. In such cases the application must adapt to changing resources. For example, a video streaming application would decrease the resolution or frame rate if the bandwidth decreases. Such applications are called *reactive*. Other applications can neither guarantee QoS properties nor can they adapt in any way. For example, typical P2P applications such as file sharing solutions cannot guarantee any throughput. And there is no way of compensating low bandwidth. The application can just monitor the QoS it perceives and tell the user how long the download will approximately take or it can abort a download if the monitored throughput is below a threshold. Hence, QoS-aware distributed applications can be pro-active, reactive, or just monitoring.

A very special kind of QoS-enabled applications are embedded real-time systems. They are very important for devices such as mobile phones or control systems as used in machine building and robotics. These systems operate in a closed environment. The available CPU power, number of concurrently running processes, memory footprint of each process, and all other resources are known a priori. Hence, it is (in theory) possible to proof that the system operates within the demanded QoS properties. If the device fails to meet its QoS properties, it can be considered to be broken. In contrast, distributed enterprise applications operate in an open environment. This environment is less predictable. New processes can be launched at any time. This affects the available resources such as CPU power, memory, bandwidth, etc. Hence, such applications must be pro-active or at least reactive to deal with their changing environment. This thesis does not focus on embedded real-time systems. However, a short overview of modeling approaches can be found in chapter 10 and chapter 15.

Another difference between embedded real-time systems and enterprise applications is the *granularity* of QoS properties. In embedded systems the runtime of a single method invocation matters. In enterprise applications performance is usually measured at the granularity of tasks instead of single method invocations. A task can involve a large set of method invocations. It does not matter how long a single method invocation runs. The important question is how long the execution of the entire task lasts. Therefore, the presented PIM modeling language will provide means for modeling such tasks and their QoS properties.

9.1 Overview

Part III is organized as follows. The next chapter discusses foundations and related work. This covers a more detailed explanation of the term QoS and the associated terminology. Furthermore, existing approaches to QoS specification are presented. This includes textual specification languages and model-based approaches. An overview of QoS-aware middleware including research projects and industry products close the chapter.

Chapter 11 presents the PIM modeling language used to model QoS-enabled applications in a platform independent way. The chapter explains how QoS contracts can be attached to a component-based design. Furthermore, dependencies between QoS contracts and the concepts of tasks are discussed. The chapter closes with a comparison between the presented modeling language and the most prominent QoS specification language QML.

Chapter 12 shows examples of platform specific models for QoS-enabled applications. The chapter starts with a discussion of deployment diagrams and how they are affected by QoS. The next four sections show PSMs for different QoS-enabled middleware platforms.

The last chapter explains how the tool chain presented in chapter 8 can be utilized to transform a PIM as shown in chapter 11 into their counterparts as illustrated by chapter 12. The chapter starts with an explanation of the term QoS-aspect and shows how QoS aspects can be modeled.

The following sections show how a Kafka transformation can be deduced from such a QoS aspect. The chapter closes with a discussion of multi-category QoS and the implications for model transformation.

Chapter 10

Foundations and Related Work

10.1 Component Contracts

The platform independent modeling language presented in this thesis builds on the concept of components. The PIM shows the components that constitute the application and details their interaction. A component is – in contrast to a class – well encapsulated [Szy02]. Components make their provided and required interfaces explicit. In contrast, classes in object-oriented languages make only their provided interface explicit. The interfaces required by the environment are not made explicit.

In the context of QoS-aware applications it is not enough to specify only interfaces using methods, their parameters, types, and attributes. These syntactical interfaces do not describe how the methods behave and perform. Hence, the term contract is introduced to denote that there are other properties (synchronization, behavior, and QoS) that must be specified. [BJPW99] defines four levels of contracts:

- syntactic contracts,
- synchronization contracts,
- behavioral contracts, and
- QoS contracts.

10.1.1 Syntactic Contracts

Syntactic contracts define which messages a component accepts and which reply-messages it will send in return. In object-oriented design syntactic contracts are expressed in terms of methods and attributes that are grouped to interfaces. Syntactic contracts can be easily expressed in the UML using class diagrams.

10.1.2 Synchronization Contracts

Synchronization contracts define in which order a client may send certain messages to a port and when return-messages have to be expected. In UML a hierarchical message sequence chart (HMSC) can be utilized to model synchronization contracts. HMSCs describe the ordering of message calls. They do not describe the effect of the message calls.

Synchronization contracts can be parameterized. For example an interaction with a query engine involves one query and the retrieval of n query results. Here, n is the parameter. If a time critical QoS contract is attached to this task, it is utterly important to determine a value for n. Only if this parameter is known it will be possible to allocate resources for the accomplishment of the task.

10.1.3 Behavioral Contracts

Behavioral contracts define the effect caused by a message sent from a client to a port. In UML behavioral contracts are specified in terms of pre-conditions, post-conditions, and invariants. Pre-conditions are utilized to describe a certain constraint that has to be true before the message is sent. Post-conditions are constraints that will be satisfied after the message has been processed. Finally, invariants describe constraints that are satisfied before the message is sent and after the message has been processed.

10.1.4 QoS Contracts

A QoS contract specifies the qualitative or extra-functional properties of a service. The term *QoS* originates from the networking area where it covers bandwidth, jitter, delay, response time, concurrent calls, and so on. The term *extra-functional* covers additional properties such as security, transaction, accounting, etc. The special fact about QoS contracts is that the quality of a service can be measured in an almost *analog* way, with the obvious limitation that today's computing hardware is entirely digital, hence, the analog values are always quantified. The bandwidth can be measured in MByte/sec, response time in milliseconds, etc. In contrast, extra-functional properties such as security and transaction are hardly measurable at runtime.

QoS contracts are usually parameterized - similar to synchronization contracts. In the case of an availability contract the expected uptime percentage is one possible parameter. In the case of security, the key length or the encryption algorithm is an example for parameterization. Existing approaches for the specification of QoS contracts are discussed in section 10.2.

10.1.5 UML Components

The UML provides concepts for the modeling of component-based systems. However, the component support of UML 1.4 is not well suited for contracts. UML 1.4 components do not have explicit connections – i.e. a direct relationship between two components. It is not possible to model that component A interacts with component B via some interface. Instead, a component can only specify that it either realizes or requires a certain interface. If multiple components provide the same interface, it is no longer possible to determine which components interact. UML 1.4 components are bound to types. They are not connected directly.

The first problem of this component model is that the model transformation does not have sufficient information for the generation of the QoS-agnostic PSM, because the relationship between the different components is not precisely modeled. Two components can accidentally provide, respectively require, the same interface. This does in no way imply that these

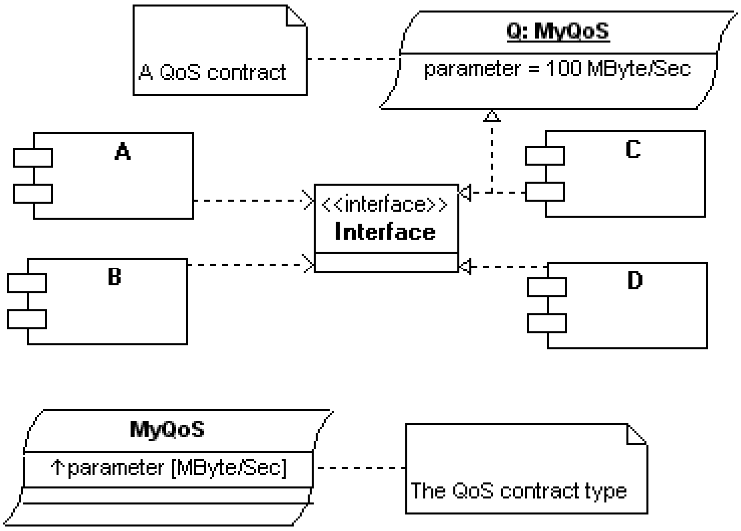


Figure 10.1: UML 1.4 components

components are supposed to communicate. They could even belong to two totally different subsystems. The second problem is that a component in such a model is doomed to offer the same QoS contracts to all client-components. The reason is that contracts cannot be attached to an inter-component connection, since such connections do not exist in the UML 1.4 meta model. Figure 10.1 illustrates this. Components A and B require the same interface. Components C and D offer this interface. It is, for example, not clear whether A communicates with C or D. Furthermore, C cannot offer A a different QoS contract than B.

New modeling languages such as EDOC [OMG02f] or UML2 [OMG03f] solve this problem by introducing ports and inter-port connections. A component communicates with its environment only via *ports*. A port has a type, i.e. a syntactical contract. Two ports can be directly connected. Furthermore, one provided port can serve multiple required ports, hence, a component can have multiple attached client-components. Contracts can be attached to the port-to-port connections. Thus, every client-component can access the server-component with different QoS properties. Figure 10.2 illustrates this. Therefore, the platform independent modeling language used in this thesis builds on the UML2 component meta model.

10.2 QoS Specification Languages

10.2.1 QML

QML [FK98b, FK98a, FK99] was the first general purpose QoS specification language. General purpose means that QML is not restricted to a certain QoS category. QML supports the specification of QoS contract types. An example of a contract type (taken from [FK98b]) specification is shown below.

```
type Performance = contract {
```

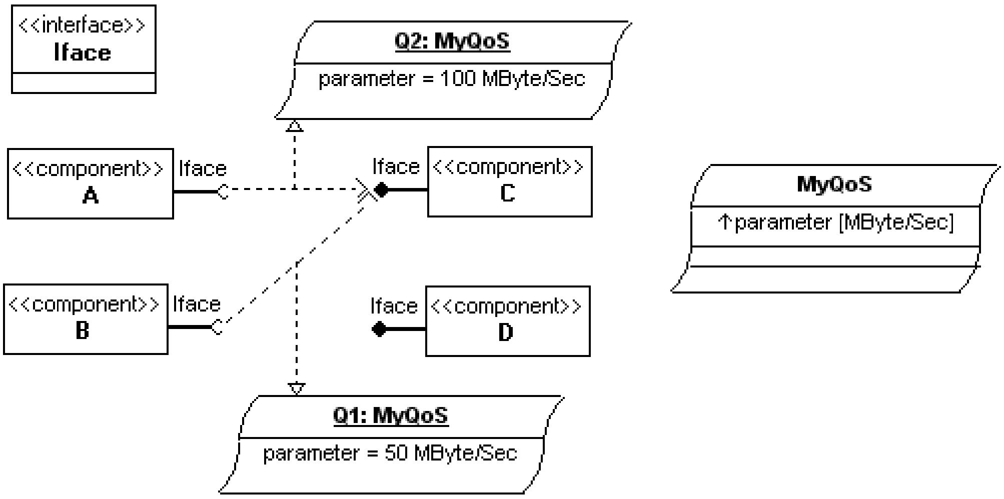


Figure 10.2: UML 2 components

```
    delay: decreasing numeric msec;
    throughput: increasing numeric mb/sec;
};

type Reliability = contract {
    numberOfFailures: decreasing numeric no/year;
    TTR: decreasing numeric sec;
    availability: increasing numeric;
};
```

The contract type named `Performance` features two so-called QoS dimensions: `delay` and `throughput`. Each dimension has name, direction, value type, and physical unit. The value type may either be numeric or an (ordered or partially ordered) enumeration. The direction determines whether higher values yield a better quality (`increasing`) or not (`decreasing`). A QML contract is built on top of a contract type as shown by the following example.

```
contract1 = Performance contract {
    delay < 40 msec
};

contract2 = Performance contract {
    delay {
        percentile 50 < 10 msec;
        percentile 80 < 20 msec;
        percentile 100 < 40 msec;
        mean < 15 msec;
    };
};
```

Equations, inequations and a predefined set of statistical functions can be used to constrain the QoS dimensions. In the above example the QoS dimension `delay` is constrained. In the first case, an upper bound is given. In the second case a statistical distribution and a mean value is demanded. The terminology used in QML is questionable. A QML-style contract is not an instance of a QML-style contract type. Instead, a QML-style contract is more kind of a specialization of a contract type with additional constraints. In the above example, a constraint expresses that the average value of the QoS dimension `delay` must be below 15 msec. Therefore, the terms *contract type* and *contract* are misleading because the naming convention implies that both terms have a *instance-of* relationship.

QML must bind the QoS contracts to interfaces or methods. This is achieved with so called profiles. A profile can bind a contract to the entire interface. This is the case for `contract1` in the following example. Alternatively, a contract can be bound to single method. This is the case for the method `bar()` in the example below.

```
interface ISomething {
    void foo();
    void bar();
};

PerformanceProfile for ISomething = profile {
    require contract1;
    from bar require contract2;
};
```

10.2.2 CQML

CQML [Aag01] is a textual QoS specification language, which resembles many ideas of QML and adapts them to the special needs of component-based systems. The most notable differences to QML is a different vocabulary, the usage of the object constraint language (OCL) [OMG03d], support for CORBA component IDL (CIDL) [OMG02b]. As in QML it is possible to describe QoS dimensions. However, in QML they are called quality characteristics.

```
quality_characteristic response_time {
    domain: numeric decreasing real milliseconds;
    mean;
    maximum;
    invariant: maximum <= 2 * mean;
}
```

Each characteristic has a domain, which specifies the possible range of values. Furthermore, a characteristic can use a set of predefined statistical functions. In the above example, the `response_time` characteristic uses a maximum and a mean. Using OCL, it is possible to specify invariants for characteristics. OCL is well suited for such purposes since it is a language free of side effects. A CQML characteristic can make use of measurands which are passed as parameters to a characteristic.

```
quality_characteristic delay( msg_sent, msg_received ) {
    domain: numeric decreasing real milliseconds;
    value: msg_received - msg_sent;
    maximum;
    invariant: msg_sent < msg_received;
}
```

In the above example, the value of the characteristic is always the time difference between the sending of a message and the reception of the response message. Hence, a CQML characteristic can be treated as a function reading metered values and yielding a resulting value and optionally some statistical values. So-called QoS statements constraint characteristics. QoS statements are comparable to QML's contracts.

```
quality fast {
     response_time < 100;
}

quality slow {
     response_time < 200;
}
```

OCL is used to express these kinds of constraints. On the above example, the response time is constrained to be smaller than 100. In this case, the quality is `fast`. Just like QML, CQML features a concept called profile. It binds several QoS statements to a set of interfaces. The following example binds two QoS statements to the provided ports of a component. The interface of the component is described using CIDL.

```
interface FooBar {
};

component MyComponent {
    provides FooBar service;
};

profile MyProfile for MyComponent {
    provides fast;
}
```

However, it is not possible to bind contracts only to a limited set of interfaces since a profile is attached to a component and not to one of its interfaces. As in QDL (see subsection 10.2.4) it is possible to define transitions between different qualities.

```
profile MyProfile for MyComponent {
    profile good { uses fast; }
    profile bad { uses slow; }
    transition bad->good: invoke_some_callback();
    transition good->bad: other_callback();
    precedence: good, bad;
}
```

The above example describes that a callback function must be invoked if the quality level changes. This way, CQML features support for adaptation. CQML features a UML profile [AE02], which has the same expressive power as the textual CQML version. Textual CQML binds quality statements via profiles to CIDL. In contrast, the UML-based version attaches quality statements to UML 1.4 components or classes modeled in UML. The drawback of the chosen approach is that the notation is ugly and not very intuitive. This is usually the case when UML stereotypes and tagged values are used excessively. The CQML profile uses 14 stereotypes and 20 tagged value definitions. 12 out of 14 stereotypes extend the UML meta class `Classifier`, hence, the stereotyped element is always displayed as a rectangular box (like a UML class) inside a UML tool. Hence, the resulting diagrams are hard to read and do not provide any advantage over the textual QML version.

10.2.3 QIDL

QIDL [BG99] is an IDL extension used by the MAQS framework [Bec01, BG97]. In contrast to QML, QIDL does not separate the definition of QoS profiles from the definition of functional interfaces. Both are specified using a single language. QIDL adds two new keywords to IDL: `qos` and `withQoS`. The first new keyword is used for the specification of a so-called `QoS interface`. The following QIDL fragment provides an example.

```
qos Replica
{
    short ServerNum;
    ...

    interface {
        State GetState();
        void SetState( in State s );
    };
};
```

The QoS interface `Replica` defines a QoS parameter. It determines the desired number of servers. Furthermore, a set of QoS-related methods are grouped to an interface. These methods are woven with the functional interface when the QoS interface is attached. The following example demonstrates this.

```
interface ISomething
{
    void foo();
};
```

```
interface ISomethingWithQoS : ISomething withQoS Replica
{
};
```

In this example, the C++ `ISomethingWithQoS` interface generated by the QIDL compiler would consist of three methods: one derived from `ISomething` and two QoS-related methods derived from `Replica`. This shows that QIDL is very close to the realization domain. A QoS interface is highly dependent on the QoS mechanism used to realize a certain QoS. For example, the methods `GetState` and `SetState` are used by the QoS mechanisms to clone server objects. In the application domain such realization details would not be visible. QIDL does not foster a strict separation between QoS specification and QoS realization. In the context of a model-driven approach this is not necessarily negative. The PIM is used to specify the QoS properties independent of their realization. The middleware and its associated tools and languages do not need to uphold this abstraction, since the PIM does already provide this abstraction level.

10.2.4 QDL

QDL [LBS+98, PLS+00b] is a Quality Description Language used by Quality Objects (QuO) [BBN03, ZBS97b, VZL+98], a CORBA-based framework for developing distributed applications with QoS requirements. QDL consists of a set of QoS aspect languages and CORBA IDL. IDL is used to specify interfaces. In contrast to QIDL, QuO does not extend IDL with new keywords. Instead, aspect languages define the QoS contract outside the interface definition. QDL features two aspect languages: (1) a Contract Description Language (CDL) and (2) a Structure Description language (SDL).

Specifying an interface with QDL starts with standard IDL. The following example shows, that IDL is used to specify the functional interfaces and interfaces used for QoS. In this example, the interface `client_expectations` is used by the QoS contract while `ISomething` is the functional interface.

```
interface ISomething {
    void foo();
    void bar();
};

interface client_expectations
{
    int requested;
};

interface client_callback
{
    toolow();
    normal();
};
```

CDL is used to specify QoS contract types and so-called *regions*. CDL-style contracts are defined independently of any functional interface. A contract defines a set of *negotiated regions* and *reality regions*. As the name indicates, the first kind of region is determined through contract negotiation. The example below specifies two regions: `Low_Cost` and `Available`. Which one applies at runtime is determined via the `client_expectations` interface defined above. Each negotiated region can contain a set of reality regions. Which reality region applies is determined via measuring. Hence, the negotiated regions are selected during negotiation, while reality regions can be switched any time. The `transitions` statement of CDL describes what happens if reality regions are switched. In the example, the client is notified via its callback interface.

```
contract Replication is ...
  negotiated regions are
    region Low_Cost:
      when client_expectations.requested = 1 =>
      ...
    region Available:
      when client_expectations.requested > 1 =>
        reality regions are
          region Too_Low : when measured <= 1
          region Normal : when measured < 1
          transitions are
            transition any->Too_Low: client_callback.toolow();
            transition any->Normal: client_callback.normal();
  ...
end Replication;
```

The SDL is used to define how calls are handled with respect to the current region. The following example shows that a call can either be forwarded to a single server object or to a group of server objects. If the QoS contract is broken (i.e. the `Available.TooLow` region applies), an exception is thrown.

```
delegate behavior for ISomething and Replication is
    obj: bind ISomething with name SingleObject;
    group: bind ISomething with ...;

    call foo:
      region Available.Normal:
        pass to group;
      region Low_Cost.Normal:
        pass to obj;
      region Available.TooLow:
        throw AvailabilityDegraded;
    return foo:
        pass_through;
end degelgate behavior;
```

In contrast to QML, QDL does not make QoS dimensions explicit. They are implicitly defined by IDL interfaces. Therefore, QDL is closer to the realization domain than the application domain. In this aspect, QDL has similarities with QIDL. In contrast to QIDL, QDL uses aspect languages, i.e. the IDL itself remains unchanged. QDL and CQML share the idea of transitions between different QoS-levels (regions in QDL parlance). A specialty of QDL is the SDL. It is used to describe *how* the QoS contract is realized. In so far, QDL is more than just a QoS specification language since is concerned with implementation specific decisions, too. However, it is not clear why SDL is called *structure* description language. SDL is used to describe which QoS mechanisms are used for which region, i.e. SDL is more kind of a *behavior* description language.

10.3 QoS-enabled Middleware

Classical middleware supports the interface specification, implementation, and execution of a distributed application. Middleware shields the developer – at least partially – from the implications of distribution. In an ideal case, a developer can implement a distributed application without any knowledge of distribution. This is called distribution transparency. QoS-aware middleware enables application to specify their QoS properties and to handle QoS provisioning at runtime. For example, components can benefit from security features or load balancing that is realized with the help of the middleware.

In order to be QoS-aware a middleware must at least feature so-called *QoS mechanisms.* A QoS mechanism is used to implement a certain QoS category. The mechanism influences the message exchange between distributed objects. Furthermore, it may influence the activation, deactivation, migration, and replication of objects. Since these activities belong to a domain covered by the middleware, QoS mechanisms are usually very tightly integrated with the middleware. Some QoS platforms allow developers to add new QoS mechanisms [UWGB03]. Therefore, they allow to dynamically plugin the QoS mechanisms into the communication path of the middleware. Such products are called *generic QoS* platforms. Others support only a built-in set of mechanisms. The combination of different QoS mechanisms is very difficult since they can interfere [UWG03]. Middleware products addressing this problem support *multi category QoS.* Other platforms avoid the problem of QoS mechanisms interference. They feature only *single category QoS*, i.e. only one QoS category per object.

QoS-enabled middleware products usually use a QoS specification language [ZBS97b, PLS$^+$00b, Bec01]. This language is used to specify *QoS contracts.* A QoS contract details which QoS properties an object supports. This contract is input to a *QoS strategy.* The purpose of the strategy is to utilize and configure the QoS mechanisms to provide the specified QoS properties at runtime. QoS contracts can either be specified during development. In this case they are hard coded. Additionally, a middleware can support *QoS negotiation.* This allows two objects to negotiate QoS contracts at runtime.

Older middleware platforms build on object-oriented principles. Objects can be very fine granular and they are not well encapsulated. Objects only define their provided service but not the services they require from their environment. This limits the possibility of reuse. Therefore, new middleware products build on the concept of *components* [Szy02]. Components provide better encapsulation and foster reuse. New QoS-aware middleware products are built on component-oriented middleware such as CORBA CCM [OMG02b]. QoS-aware components developed with component-oriented middleware specify their *provided* and their *required* QoS properties. In contrast, QoS-aware objects running on top of an object-oriented middleware such as MAQS [Bec01] specify only their provided QoS properties.

Most QoS-aware middleware products are build on top of an existing middleware [Bec01, Rit03, UWGB03, PSC03]. Several approaches have been developed by the scientific community. Most of them are based on CORBA since this used to be the dominant middleware when these projects started. Other platforms such as .NET have been extended lately to incorporate QoS support [UWGB03, UWG03, WUG03b] explicitly in the .NET Remoting middleware [Ram02]. Companies have added QoS related products to their middleware and operating system products, too. For example, Microsoft provides the COM+ services [Ise00] for its Windows OS. COM+ supports a fixed set of mechanisms, among them security, events, object pooling, message queuing, and component load balancing [RAC$^+$02] that can be used to realize several QoS categories.

The following sections discuss a selection of QoS-aware middleware products. The focus is on the implications for the model transformation. From the viewpoint of the transformation, middleware products can be categorized along several lines.

- *Multi Category QoS:* Some platforms support more than one QoS category for one component at the same time. If the target platform does not support this, some PIMs cannot be mapped to this platform. This can happen if multiple QoS contract types are attached to one port.

- *Generic QoS:* MAQS [Bec01] and DotQoS [UWGB03] potentially support a very large range of QoS categories. Almost every QoS contract type that can be modeled in a PIM can be realized with such a middleware. The model transformation just needs to add the required QoS mechanisms since generic QoS middleware provides only the hooks for QoS mechanisms. In contrast, non-generic QoS middleware restricts the PIM to those QoS contract types that are supported by the middleware.

- *Contract notion:* Most middleware products originating in the research community support the notion of a QoS contract. A QoS contract specifies which QoS properties a component provides and requires. The model transformation must map the contracts found in the PIM to contracts supported by the middleware. If the middleware does not support contracts, the model transformation must select and configure the appropriate QoS mechanisms that are able to realize the contract.

- *Contract negotiation:* The PIM can define concrete QoS properties or it can just enumerate the supported QoS category. In the second case, concrete QoS properties are determined at runtime via contract negotiation. If the targeted middleware does not support contract negotiation, the model transformation must either add negotiation support on top of the middleware or it must advice the developer to specify concrete QoS properties of all QoS-aware components.

- *Client-specific contracts:* A QoS-aware component can either provide the same QoS properties to all of its clients or it can provide different QoS properties to different clients. If the first case applies, the model transformation cannot realize all possible PIMs with the help of this middleware. A component must not have multiple clients with different QoS demands if the middleware supports only the same QoS properties for all clients.

- *Reflective approach:* DotQoS relies heavily on reflection and interception of messages. This allows QoS mechanisms to hook themselves up to the message path at runtime. Thus, the separation between QoS-related code and QoS-agnostic code can be kept up even during implementation. In contrast, MAQS is built on an AOP-based approach. QoS-related code is injected at design time in the stubs and skeletons generated from the interface definition language. Hence, the separation is lost – at least partially – at the source code level. A more detailed discussion of reflective middleware can be found in [KCCB02].

- *Support for tasks:* Some middleware products allow QoS only on a per interface or per method basis. As discussed in section 11.4 it can be very useful to apply certain QoS properties to a context. For example, a web shop requires authentication only when a customer starts adding items to his virtual shopping basket. Hence, the same functionality can be provided with or without QoS depending on the context. Tasks are introduced in the PIM to model this kind of context. If a middleware supports QoS only on a per interface basis, some platform independent models cannot be mapped directly onto this middleware.

10.3.1 CORBA

Several research projects extended the Common Object Request Broker Architecture (CORBA) with QoS support. The TAO project [Sch03, PSC03] extended CORBA with real-time capabilities. In the meantime, the OMG has standardized real-time extensions for CORBA [OMG03c]. This standard is supported by the TAO Object Request Broker (ORB). TAO is aware of the different priorities a thread can have. During remote invocations the client thread priority can be propagated to the server. Based on this priority, the server schedules incoming message calls to meet their timing constraints. On the network level, protocols such as DiffServ [IET98] are used deliver messages in time. A resource planner is available that can either statically or dynamically [GLS01] determine whether a demanded real-time guarantee can be met by the system or not. In summary, TAO supports one QoS category (real-time) and has a notion of client-specific timing contracts. If dynamic scheduling is used, it is even possible to negotiate contracts at runtime.

Another CORBA-based project is QuO (Quality Objects) [BBN03, ZBS97b, VZL+98] from BBN Technologies. QuO has been developed mainly for the military sector. It supports QoS categories which are important for battle field technology, such as real-time and availability [RBC+03]. A focus of QuO is on adaptation. Its QoS specification language QDL (see subsection 10.2.4) describes QoS regions and when the application should switch between the different regions. On the implementation side, QuO inserts delegates between the client and the client-side ORB. These delegates consult contract objects to determine the current QoS region. Based on this, the delegate uses an adequate QoS mechanism. For example, if a large image of the combat field is to be retrieved, it can be split into several tiles. The tiles are requested subsequently and their resolution is adapted to the currently available network resources. So-called system condition objects inside QuO measure the provided QoS level and trigger a change of QoS regions if required. From the client perspective, the entire QoS-related processing is hidden behind the delegate objects. A specialty of QuO is that the QDL files can contain source code. A QDL file describes, for example, which actions an application should take if QoS region are switched. In summary, QuO offers several multi category QoS. In theory, QuO can be extended with support for new QoS categories (i.e. generic QoS support) via new contract objects, system condition objects, and delegates. However, this will require changes to the QDL code generators.

MAQS [BG97, Bec01] is a QoS extension of the MICO [RP03] CORBA implementation. The goal of MAQS is to support generic QoS management on standard object-oriented middleware. MAQS builds on an aspect-oriented approach to separate the QoS mechanisms from business logic. The QoS applicable to a certain interface is defined in an extension of CORBA IDL called QIDL (see subsection 10.2.3). A QIDL file specifies QoS-specific attributes and methods, which are used to negotiate, monitor, and adapt the QoS level at runtime. The QIDL compiler extends the standard CORBA IDL compiler. Furthermore, MAQS extends the CORBA IDL to C++ mapping with QoS-related constructs. The compiler acts as an aspect weaver and combines the CORBA skeletons and stubs with QoS-related source code that supports the integration of QoS mechanisms at runtime. The QoS-related source code in the generated stubs and skeletons uses an interceptor-pattern on client side and a chain of responsibility pattern on server side. Thus, it is possible to integrate QoS mechanisms in the ORB at runtime using a reflective approach. This way, the implementation of a CORBA object is separated from a concrete QoS mechanism implementation. QoS mechanisms can be exchanged at runtime if they support the QoS-specific interface (i.e. attributes and methods) as defined in the QIDL file. MAQS has been designed as a generic QoS middleware. However, it can only handle one QoS category per object. Multi category QoS is not addressed by MAQS.

10.3.2 COM+

Microsoft's operating systems offer prefabricated solutions for some QoS-related mechanisms. These are called COM+ [Ise00]. The name implies that COM+ is an extension of COM [Box97], the dominant component model for Microsoft products. COM+ acts – conceptually – as a wrapper around COM components. COM+ offers the following services QoS-related mechanisms:

- *Transactions:* A component can participate in a distributed transaction service. This is very important for critical applications. COM+ implements a two phase commit protocol.
- *Security:* COM+ can authenticate the client invoking a COM+ component. Authentification can happen at several occasions: (i) upon the client's connection to the COM+ component, (ii) with each network package exchanged between the client and the COM+ component, or (iii) as each method is invoked. COM+ can perform authorization, too. It checks the role of the client and rejects method invocations if the role is not entitled to invoke the method. If required, COM+ can impersonate a component. That means the component is executed with all privileges and restrictions of its calling client.
- *Just-In-Time Activation:* Some components allocate many expensive resources when they are instantiated. Therefore, the activation of such components should be deferred until they are invoked for the first time. If such components are no longer used, it is be

beneficial to release their resources. JIT activation can be used to limit the resources such as memory or database connections consumed by a component.

- *Object Pooling:* If performance is critical and components have a long startup procedure, object pooling can be used. COM+ allocates a pool of COM+ components. Multiple clients can use the same instantiation of a COM+ component. This avoids the time consuming startup procedure whenever a new client requests a component. The size of the pool is adapted dependent on the number of clients.
- *Component Load Balancing:* A special version of Microsoft's server operating system is required to realize COM+ load balancing. Upon a request sent from client to a component, the component load balancer determines one machine and assigns the client to it. The COM+ component must be stateless, since subsequent requests may be redirected to different machines.
- *Message Queuing:* Sometimes a COM+ component may be unavailable. Either it has crashed or the client is currently disconnected from the network. In such situations message queuing records the messages sent from the client to a COM+ component and sends them again as soon as the COM+ component becomes available.
- *Events:* COM+ Events are an implementation of the publish-subscribe paradigm [Müh02]. They can be useful to monitor the load of a distributed system. However, COM+ Events are weak because a malicious client can block the transmission of an event.

Although COM+ is not a dedicated QoS technology such as DotQoS, Qedo, or MAQS, it can be used to implement QoS-aware applications. The most important difference is that COM+ does not support the notion of a contract. It just provides a set of QoS mechanisms. Contract negotiation, monitoring, and adaptation strategies are not covered by COM+. Microsoft's COM+ is not a generic QoS framework. Developers cannot add new QoS mechanisms. They are restricted to those listed above. COM+ is partially multi category QoS enabled. For example, security can be used in combination with other QoS mechanisms, but it is not possible to combine load balancing with transactions. COM+ in combination with .NET is highly reflective. Custom attributes attached as meta information to a class determine the QoS mechanism that has to be applied to the component implemented by this class. This is detailed in section 12.2. COM+ does not support tasks. With the exception of security, the QoS mechanisms apply to every method, event, or message supported by a component.

10.3.3 ASP.NET

ASP.NET is an acronym for active server pages in the .NET framework. An active server page is an HTML page that is dynamically constructed in the web server when the page is requested by the client. The concept is similar to Java's JSP technology [Sun03]. ASP.NET applications can use some QoS mechanisms that are built into the ASP framework and the web server that hosts the active server pages. ASP.NET supports security, i.e. authentication and authorization, transactional behavior, and load balancing. QoS is not a first class concept in ASP. The concept of a QoS contract does not exist. Instead, some modifications in XML configuration files, custom attributes, and the implementation of specific interfaces can turn on some mechanisms that can be used to fulfill a QoS contract.

COM+ configures QoS mechanisms either programmatically or via custom attributes attached to classes. In addition, ASP.NET uses XML configuration files to configure QoS mechanisms. This implies that the model transformation must be able to map PIM contracts to the appropriate XML tags. Therefore, XML tags must be part of the PSM.

10.3.4 DotQoS

DotQoS [UWGB03, UWG03, WUG03b] is a QoS-aware middleware that is based on .NET and .NET Remoting. The .NET framework and related Microsoft technologies provide several approaches for the development of distributed systems. The web-based ASP.NET and COM+ have already been discussed. From a design point of view .NET Remoting is close to CORBA. Client and server communicate via a proxy and a transport channel. The major difference in comparison to CORBA is the extensibility and the reflective approach in .NET Remoting. Applications can modify and monitor almost the entire message path, ranging from the proxy, over formatters, to the transport channel [UWGB03, UWG03]. .NET Remoting relies heavily on reflection. When a message path for a server-side object is created, .NET Remoting searches for custom attributes attached to the class of the object. This custom attribute can be used to influence the message path. For example, messages can be encrypted or compressed [Ram02] before they are put on the transport channel. Or another transport channel, for example one with bandwidth reservation [Kuh03], can be used. Figure 10.3 shows the .NET Remoting stack. The message path is formed by message sinks and a channel. More information about the .NET Remoting stack can be found in [Ram02].

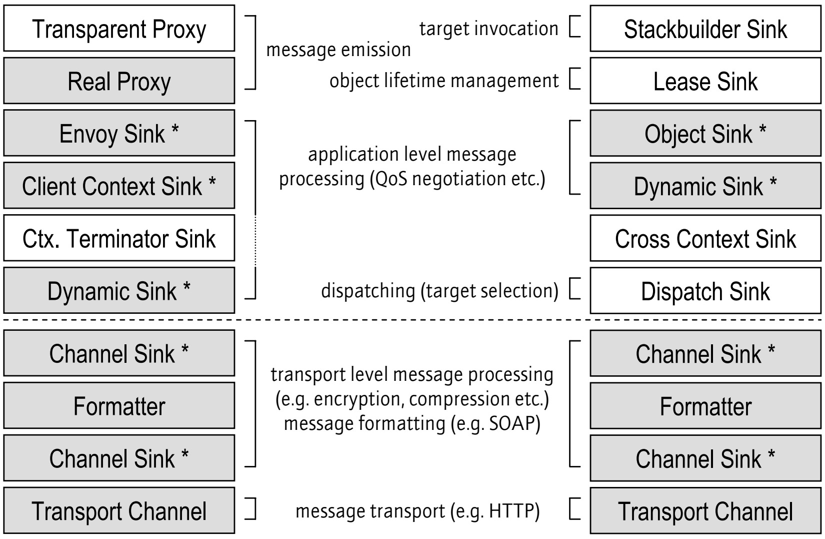


Figure 10.3: .NET Remoting stack

DotQoS is a layer on top of .NET Remoting. DotQoS handles the installation of QoS mechanisms – *sinks* in .NET Remoting parlance – and QoS-aware transport channels transparently for the application developer. The developer just attaches custom attributes to his server-side classes. These custom attributes are inspected by DotQoS at runtime. It adapts the message path, cares about contract negotiation, and contract monitoring.

.NET Remoting is not able to modify a message path after it has been created. DotQoS overcomes this problem by introducing special sinks (called QoS mechanism sinks) that forward each message to the appropriate QoS mechanism. This is depicted in Figure 10.4. More information about DotQoS can be found in [UWGB03, UWG03, WUG03b].

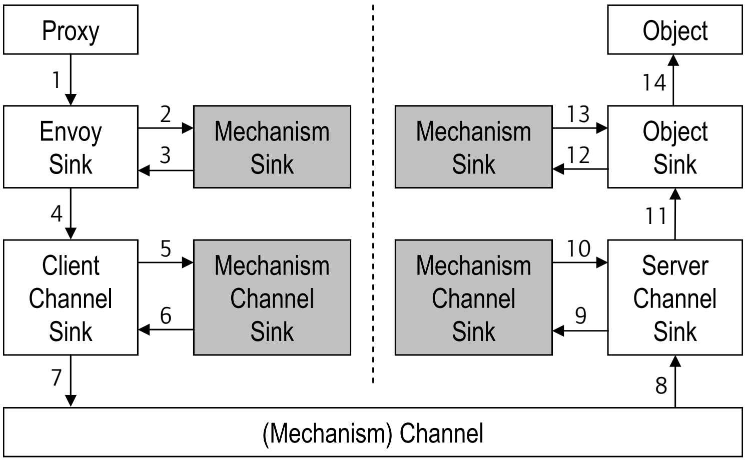


Figure 10.4: Client- (left) and server-side (right) message path in DotQoS

The following example shows a component that implements the `IFoo` interface. For all methods of this interface, the QoS category `QoSBar` is implemented.

```
public interface IFoo
{
    void Foo();
}

[DotQoS.QoSContextAttribute("TestBaseDefinitions.MyComponent")]
[DotQoS.QoSContractClass(typeof(QoSBar), typeof(IFoo))]
public abstract class MyComponent : DotQoS.RemoteObject, IFoo
{
    public abstract void Foo();
    protected MyComponent(bool register)
              : base(register) {}
}
```

From the viewpoint of a model transformation, DotQoS and COM+ have some similarities. Both rely on custom attributes. However, DotQoS is a QoS middleware with support for contracts and arbitrary QoS mechanisms. In contrast, COM+ features only a fixed number of QoS mechanisms and does not implement the notion of contracts. DotQoS implements QoS contract types as separate classes.

```
[Serializable]
public class QoSBar : QoSCategorySchemeBase
{
    [QoSDimension(Increasing,BytePerSec)]
    public int Bar
    {
        get
        {
            return (int)this.Parameters["bar"];
        }
        set
        {
            this.Parameters["bar"] = value;
        }
    }
}
```

Instances of `QoSBar` describe the QoS properties of a service. The class is tagged as serializable, since it is transferred between client and server during contract negotiation.

Contract negotiation and activation is implemented in DotQoS by a so-called frame contract. This frame contract resides on the server side. Each client of a component has its own frame contract. Hence, different clients of the same component can negotiate different QoS properties. The following example shows how a frame contract is configured and activated.

```
  // get hold of the frame contract
  DotQoS.FrameContract contract = component.CreateFrameContract(
    DotQoS.ContractServices.DefaultObserver);

  // set the desired QoS properties
  QoSBar bar = new QoSBar();
  // setting the QoS parameter(s)
  bar.Bar = 1;

  // specify QoS for IFoo port
  // port is selected with interface name, assembly, and version
  contract.SetQoSParameters(
    typeof(IFoo).AssemblyQualifiedName, bar);

  // activate contract (negotiation, resource allocation)
  contract.Activate();
```

In contrast to COM+, DotQoS can be used to implement the task concept. COM+ applies its QoS mechanisms to every invocation, disregarding of the invoked method and any context. DotQoS handles QoS more fine granular. QoS categories are applied to interfaces. Different interfaces of one component can be subject to different QoS categories. Furthermore, special message sinks can select the QoS categories applicable to a certain message depending on a context.

10.3.5 Qedo

Qedo adds QoS support to CORBA CCM, which is a component model for CORBA. Similar to CORBA, CCM uses an interface definition language to describe the interfaces of the component. The following CORBA 3 IDL example is taken from [OMG02b]. CORBA 3 IDL extends the CORBA 2.x IDL with constructs for components.

```
module LooneyToons
{
    interface Bird
    {
        void fly (in long how_long);
    };
    interface Cat
    {
        void eat (in Bird lunch);
    };
    component Toon
    {
        provides Bird tweety;
        provides Cat sylvester;
    };
    home ToonTown manages Toon {};
};
```

The IDL module describes two interfaces `Bird` and `Cat`. The component `Toon` offers both interfaces under the name `tweety` and `sylvester`. Each CORBA CCM component has a home that is responsible for the life cycle management and configuration of a component. In the above example the home is called `ToonTown` and is responsible for the component `Toon`.

Mapping this IDL to source code is a non-trivial process especially since components can have a persistent state. Persistency in turn can be implemented with a variety of database products. A complete overview of the classes involved in the `Toon` component can be found in figure 3-2 of [OMG02b]. The CORBA CCM standard facilitates an additional specification language called CIDL. It is a mixture of IDL and the Persistent State Definition Language (PSDL). The CIDL configures the code generation. For example, it determines class names, patterns (for example delegation), and binding to databases. The following CIDL example taken from [OMG02b] describes an implementation binding for the above IDL.

```
import ::LooneyToons;

module MerryMelodies {
    composition session ToonImpl {
        home executor ToonTownImpl {
            implements LooneyToons::ToonTown;
            manages ToonSessionImpl;
        };
    };
};
```

The suffixes `Impl` indicate that these names denote names of implementation classes. This CIDL specification causes the generation of the skeleton for the *component executor* `ToonSessionImpl`, and the complete implementation of the *home executor* `ToonTownImpl`.

Usually, IDL and CIDL use a textual notation. However, the OMG has standardized a meta model for both languages (chapter 8 in [OMG02b]). The textual notation can be treated as a special notation of a model based on this meta model. Figure 10.5 shows an excerpt of the BaseIDL package that defines the IDL.

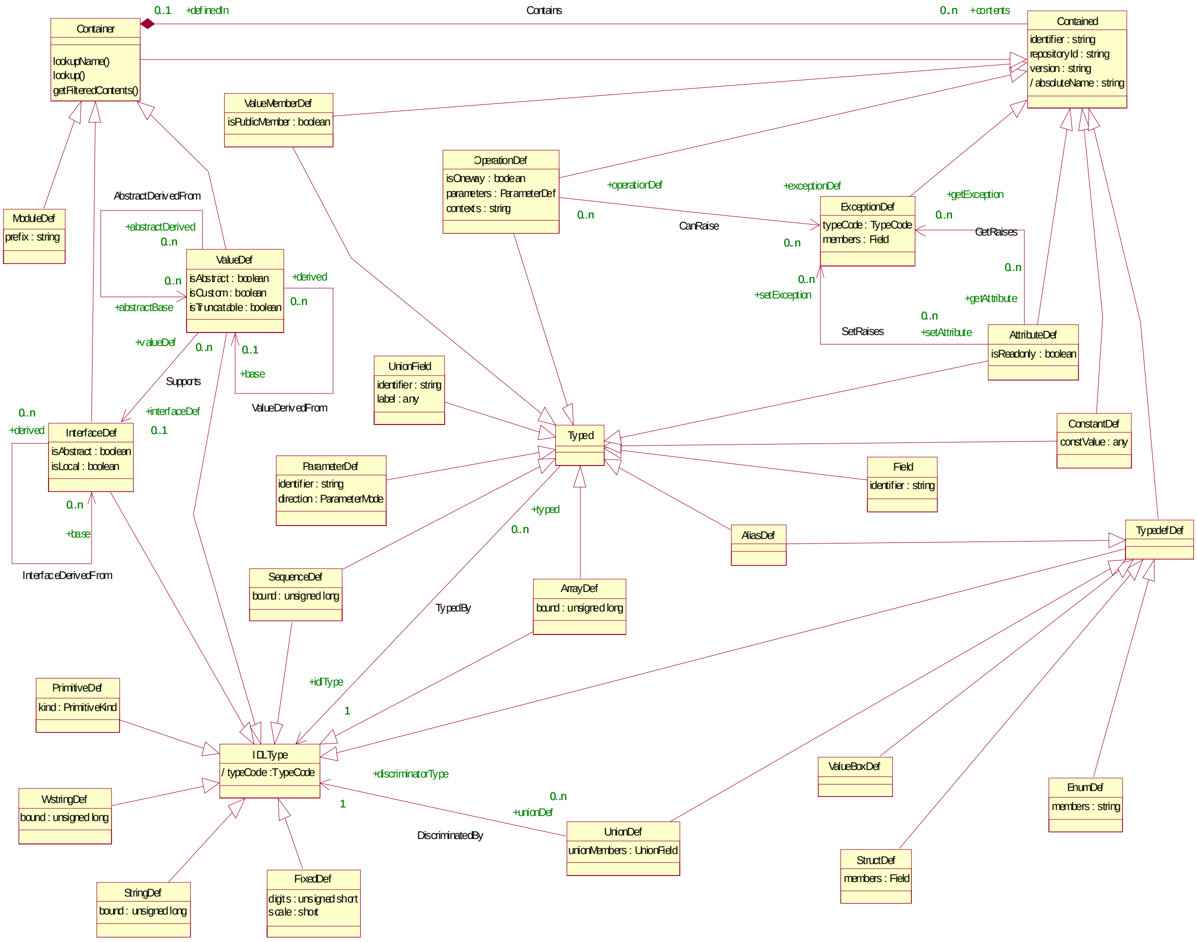


Figure 10.5: Excerpt of the IDL meta model, taken from [OMG02b]

For example, the meta class `InterfaceDef` is used to define an interface. Interfaces are `Containers`. They can contain `Contained` elements such as `AttributeDef` or `OperationDef`. For most constructs the mapping between meta model and textual notation are obvious.

CORBA CCM is not a QoS-enabled middleware. However, research groups are currently expanding their expertise on QoS-enabled CORBA 2.x platforms to CORBA CCM [Rit03, WSKP01]. Qedo is an experimental CORBA CCM implementation implemented at Fraunhofer Fokus. Qedo incorporates QoS support. Further details on the implementation strategy can be found in [RBUW03]. Figure 10.6 shows Qedo's QoS meta model.

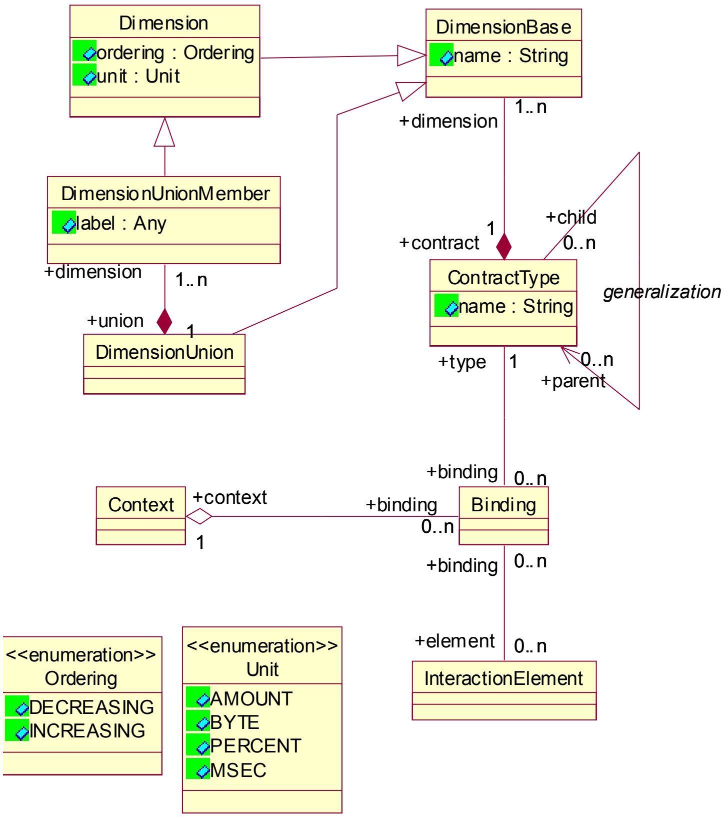


Figure 10.6: Qedo QoS meta model, taken from [RBUW03]

A `ContractType` consists of `Dimensions` and unions of `Dimensions`. Each dimension has a unit, ordering information – i.e. decreasing or increasing – and a name. Contract types are bound to interaction elements – i.e. operations, attributes or an interface – via a `Binding`. A binding belongs to a certain context. In case of the IDL a component is a valid context. A binding can express, for example, that the `fly` operation of interface `Bird` is bound to a certain QoS contract type when it is invoked on the `Toon` component (see IDL example above).

The QoS meta model shown in Figure 10.6 must be merged with the IDL and CIDL meta model. Figure 10.7 illustrates this. The constructs `InterfaceDef`, `AttributeDef` and `OperationDef` from the BaseIDL package inherit from the abstract meta class `InteractionElement`. Thus, single attributes (its setter and getter methods), operations or an entire IDL interface can participate in a QoS binding.

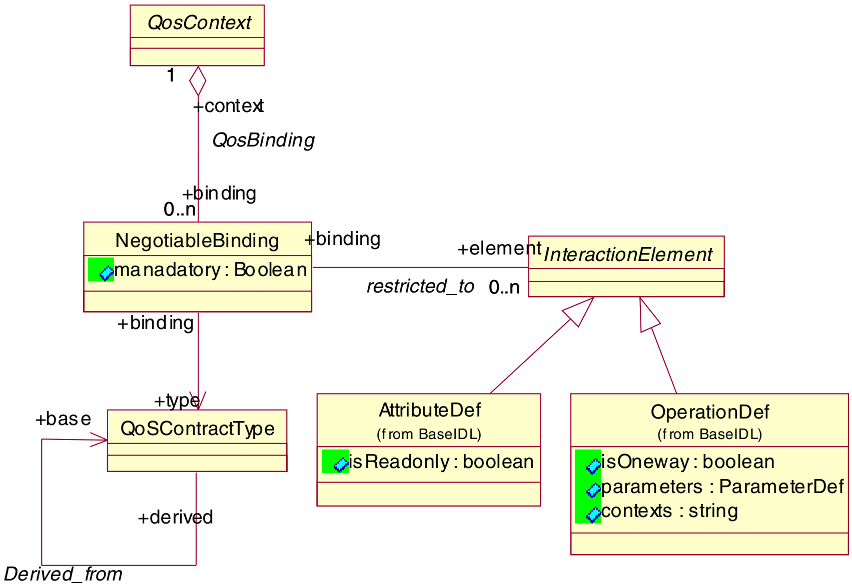


Figure 10.7: Binding QoS meta model to BaseIDL, taken from [RBUW03]

Qedo extends the IDL/CIDL meta model with QoS concepts. A textual notation for the extended IDL/CIDL is not available yet.

Chapter 11

Platform Independent Model

This chapter discusses how to a build platform independent model (PIM) of a QoS-aware application. The overall goal of a PIM is to model the application independently of its concrete realization. In the case of QoS-aware applications a PIM declares which QoS properties the application adheres to. PIMs can hide the heterogeneity of platforms and technologies used in large scale distributed systems. For example, the QoS properties of a web service and those of a CORBA application are modeled in the same way in the PIM although their realization is entirely different.

11.1 Platform Independent Modeling Language

Modeling QoS-enabled applications in a platform independent manner requires a platform independent modeling language (PIQML) enriched with QoS concepts. As detailed in section 4.1 a modeling language requires:

- an abstract syntax,
- well-formedness rules,
- notation, and
- semantics.

The abstract syntax is expressed as a meta model. One possibility is to create a new meta model from scratch. However, meta models for important concepts such as components, message sequence diagrams, and interfaces have already been developed. Therefore, the PIQML presented in this thesis is an extension of the UML 1.4 meta model. A future updated version could be ported to the new UML 2.0 meta model, but at the time of writing of this thesis the UML 2.0 is not yet a formal OMG standard. The UML 1.4 misses three important concepts. First, it does not provide means for modeling QoS contracts. These concepts have been added to the UML meta model as described in section 11.2. Second, the UML 1.4 components are not sufficient for the modeling of contract-aware components as explained in section 10.1. Therefore, the component meta model of UML2 [OMG03f] has been back-ported to the UML 1.4 meta model. Third, the UML 1.4 has insufficient support for message sequence diagrams. The old sequence diagrams have an insufficient meta model and modeling of branches, loops, parallelism, etc. is very complicated. Hence, the Hierarchical Message Sequence Chart (HMSC) meta model of the same UML 2.0 draft has been back-ported to UML 1.4. Figure 11.1 shows this graphically.

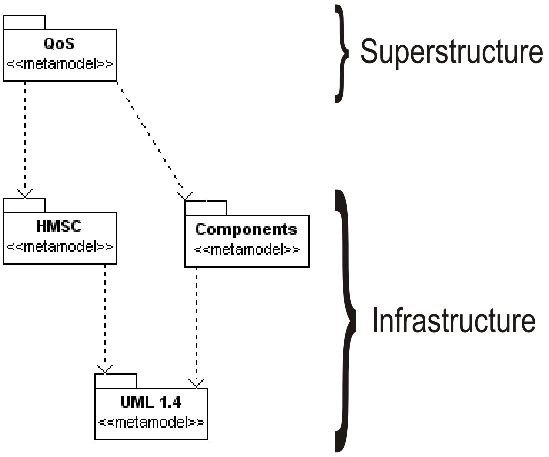


Figure 11.1: Dependencies between the meta model packages

The figure distinguishes between an infrastructure and a superstructure. The infrastructure is a compilation of existing meta models that form the basis of the PIQML. The meta model for QoS is built on top of this infrastructure. UML 2.0 itself follows a similar approach. The UML 2.0 meta model is divided into an infrastructure part [OMG03e] and several superstructure parts [OMG03f]. The distinction between an infrastructure and a superstructure is especially useful in the context of the MDA. New problem domains require new PIM meta models. New PIMs should not reinvent basic concepts. They should reuse existing and approved concepts. If all PIM meta models can build on the same infrastructure, the development of modeling and transformation tools is tremendously simplified. New PIMs could build on a thinner layer - the UML 2.0 infrastructure.

11.1.1 Components and Ports

Components provide very strict encapsulation. Ports are the only connection between the internals of a component and its environment. Therefore, it is straight forward to attach QoS properties to the ports of components. Another advantage of this approach is that it separates the QoS-related modeling elements from the business logic. The business logic is encapsulated inside the component, while the QoS properties are attached to the outside, i.e. the ports of a component. Figure 11.2 shows how components, ports, and the infrastructure meta model are connected.

The meta class for components has been named `Component2` to avoid conflicts with the UML 1.4 components which are part of the infrastructure, too. `Component2` derives from `Classifier` just as `Interface`. A component can own `Ports` because a component is a `Classifier` and classifiers can own `Features` such as `Ports`. A `Port` is typed because it is a `StructuralFeature`. An additional well-formedness rule, which is not visible in the meta model diagram, demands that this type has to be an `Interface`.

This is illustrated by Figure 11.3, which shows an instantiation of the meta model. The component in the figure owns one port which is bound to an interface. Of course, the notation

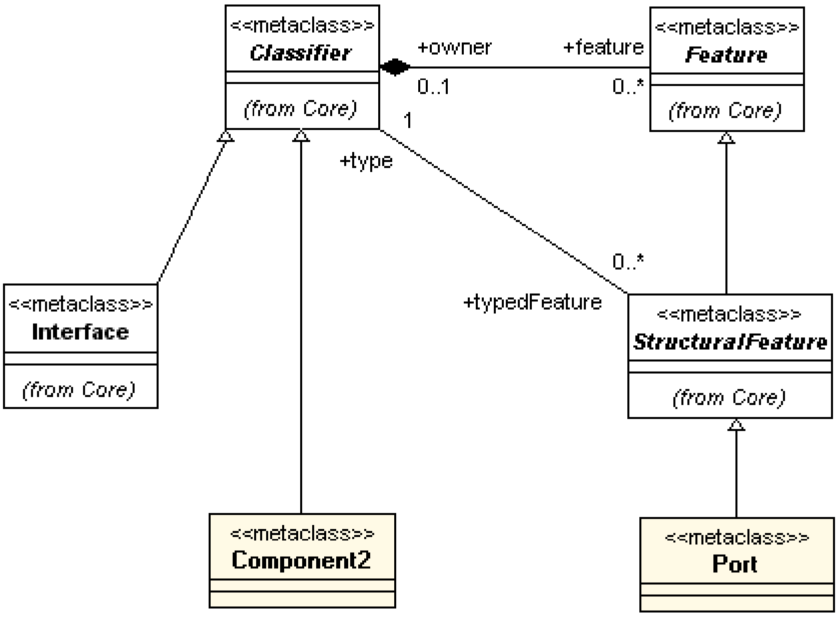


Figure 11.2: Components, ports, and an excerpt of the infrastructure meta model

is awkward. Figure 11.4 shows the same model but this time with the usual notation.

11.1.2 Contracts and Ports

Three possible ways of enriching a component with QoS properties can be envisaged. First, interfaces could be extended with QoS properties. However, a single interface might be used by several components with different QoS properties. Therefore, attaching QoS properties directly to an interface is no good choice. Second, the `Port` meta class could directly be enhanced with QoS specific concepts. However, several ports may adhere to the same QoS properties. Duplication is never a good idea. Third, interfaces, ports, and QoS properties are separate concepts. Figure 11.5 illustrates the resulting meta model.

The figure illustrates how the four levels of contracts (see section 10.1 are realized in the meta model. The first level (*syntactic contracts*) is represented by the meta class `Interface` that is attached to `Ports`. The modeling of `Interactions` allows for the specification of *synchronization contracts*. The *behavioral contracts* - usually pre-conditions, post-conditions, and invariants - are specified in UML as `Constraints`. Finally, *QoS contracts* are realized via the `QoSContractType` meta class.

The abstract meta class `ContractType` serves as a base class for the different contracts. A `ContractType` can be attached to a `Port` via a `Realization`. The realization element between contract type and port is useful to store additional information about how or when the port realizes this contract type. In an ideal world, `Interaction` should be renamed to `BehavioralContract` and `Interface` as well as `Constraint` should inherit from `ContractType`, too. This would put the meta model in line with the concept of the four levels of contracts discussed in section 10.1. However, this would involve too many changes of the UML meta model,

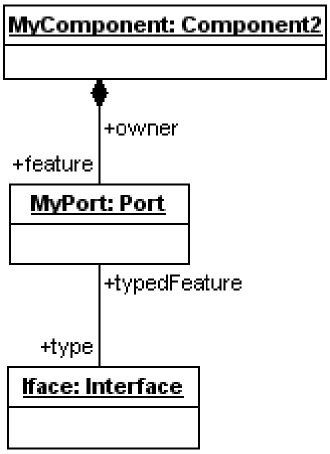


Figure 11.3: Sample instantiation of the `Component2`, `Port`, and `Interface` meta classes

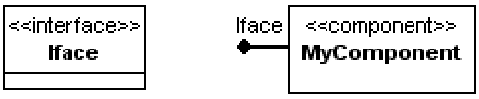


Figure 11.4: Same as Figure 11.3 but with the usual notation

i.e. the infrastructure. Therefore, the interface and constraint meta class remains unchanged. The meta class `Interaction` stems from the `HMSC` package shown in Figure 11.1 which in turn is inspired by a UML 2.0 draft. To keep consistency its name remains, too.

The PIM meta model presented in this thesis distinguishes *contract types* from *contracts*. A contract type can be compared with a class in object-oriented programming. A contract type serves as template for contracts. Contracts are an instantiation of a contract type. Therefore, the meta class `ContractType` derives from `Classifier`. UML supports the modeling of classifiers and the modeling of classifier instantiation (see subsection 11.2.6). Throughout this thesis the terms contract type and contract are treated as synonyms. An instantiation of a contract type is always called *contract instance*.

11.2 QoS Contracts

The following section is dedicated to QoS contracts. The remaining three levels of contracts are not discussed in depth. Syntactic contracts are covered by most common books on object-oriented design, e.g. [Boo93]. A more complete coverage of behavioral contracts can be found in [Mey00, Mey92]. The details of HMSCs - i.e., synchronization contracts - are discussed in the UML2 super structure [OMG03f].

To support the modeling of QoS contracts in the meta model two problems need to be solved. First of all, the concept of a QoS category has to be supported by the meta model. That means, application designers must be able to express terms such as availability, throughput, timeliness,

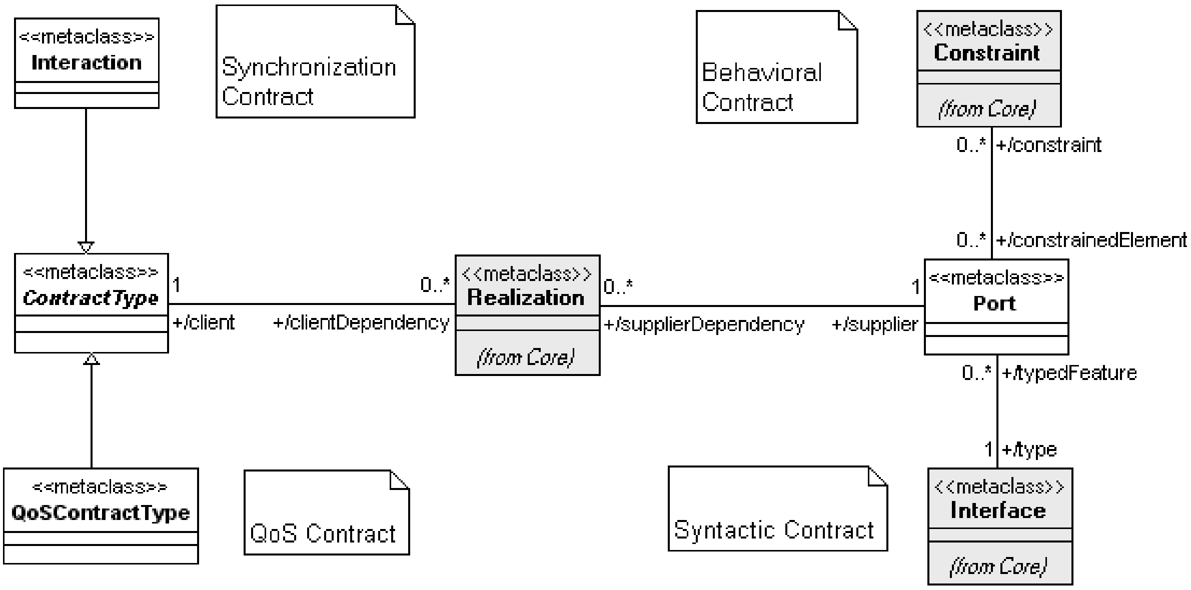


Figure 11.5: Adding contracts to ports

etc. in the model. As discussed in the following section, QoS categories can be referenced by name. Hence, their semantics are not described in the model. Instead, an external information source such as books, web pages or common knowledge is required. Another more sophisticated solution is to express the semantics of QoS categories in the model itself.

Second, the meta model must provide means for modeling constraints upon these categories. In case of a throughput contract, such a constraint limits the acceptable throughput. In this thesis, a very general approach is taken. An evaluation function (see subsection 11.2.2) is used to express the constraint.

The following discussion highlights all aspect of the QoS meta model. However, the figures only show that part of the meta model that is relevant to the current subject of discussion. The complete QoS meta model can be found in section B.3.

11.2.1 Semantics of QoS Categories

Models usually capture the structure and often the behavior of a system. However, it is hard to express semantics in a model. If it is possible to describe the semantics of a response time contract in the model, the model provides the answer to the question: What is response time? It is a philosophical question whether semantics can be absolutely described at all. Usually, the semantics of something complex is reduced to the semantics of more simple things. For example, UML state diagrams are transformed into Petri Nets [Pet81] which are conceptually clearer and well understood. This way, semantics are assigned to the state diagrams. However, this does not answer what the semantics of Petri Nets are. A Petri Net can be described by mathematic formulas. Finally, mathematic formulas reside on a set of axioms on which mathematics is built.

In order to express the semantics of QoS categories in the model the semantics have to be reduced to something that is well understood and non-ambiguous. For some QoS categories this is almost instantly the case. Terms such as authorization, authentication, indisputability, transaction, and payment are satisfactorily clear. Therefore, such contract types can use their name to assign the desired semantics. The case is different for more *analog* QoS categories which include measurements and statistics. For instance, every developer should know what response time is. Furthermore, a developer is expected to know that the average response time and its variance are usually used to calculate the quality of a service. However, the semantics of an average response time 140 ± 10 ms could be specified more precisely. The hitch is that the average response time alones does not tell how many calls have been measured to produce the average. Still, the value 140 ± 10 ms at 1000 calls does not describe precisely when the timer started and when it stopped. The response time could be time that elapses between the sending of the request message over the network and the receipt of the response message. Alternatively, the response time could be the time elapsed between the beginning of the marshalling of the request message and the end of the un-marshalling of the response message. The semantics of the average response time could be reduced to the following statements:

- The response time of a call is the time that elapses between a send event and a receive event.
- The send event occurs when the request message is send to the network driver of the operating system.
- The receive event occurs when the response message has been received from the network driver.
- The average response time is the average of the response times of n subsequent calls.

A send event and a receive event each represent one measurand. Each time the event occurs, a new value is metered. The response time of a call is a derived value of the metered values. The average response time is computed based on a sequence of such values. Hence, an evaluation function that works on the metered values can determine whether the QoS is satisfied or not. This indicates that the semantics of a QoS category can be reduced to a set of *measurands* and an *evaluation function*.

11.2.2 Evaluation Function

The evaluation function checks the delivered quality – by inspecting the metered values – and compares it with the demanded quality. The output of this function - called f from here on - is a boolean value: $true$ means the contract is obeyed, $false$ indicates that the contract has been violated. During runtime a set of measurands m_i is continuously monitored. This results in a list

$$[m_i] = m_{i_1}, \cdots, m_{i_l}$$

of metered values for measurand m_i. The evaluation function f can be defined using the measurands m_i.

Definition 11.1. A function f is an *unparameterized evaluation function* if it projects the sequences $[m_1], \cdots, [m_n]$ of measured values onto $\mathbb{B} = \{true, false\}$.

m_i are measurands, $1 \leq i \leq n$

M_i is the set of all possible metered values of m_i

$[M_i]$ is the set of all possible sequences of metered values of the measurand m_i

$[m_i] = [m_{i_1}, \cdots, m_{i_l}] \in [M_i]$ with $m_{i_j} \in M_i$ is a sequence of metered values

$f : [M_1] \times \cdots \times [M_n] \mapsto \mathbb{B}$

The delivered quality as indicated by $[m_i]$ is acceptable if f returns *true*. Otherwise the delivered QoS is unacceptable.

Expressing a valid and reasonable evaluation function requires detailed understanding of the QoS category and the measurands. It is rather unpractical if every application designer starts creating his own evaluation function, especially since a new evaluation function might require a special model transformation. Building new transformations involves extra work and costs. Therefore, existing evaluation functions should be reused if anyhow possible. Furthermore, some QoS-enabled middleware architectures do not support negotiation of functions (for example DotQoS, MAQS, and Quedo). Instead, A fixed set of values is used for contract negotiation. Consequently, f should be parameterized. This allows developers to tune f without introducing a new evaluation function. Middleware platforms that do not support the negotiation of functions will simply negotiate the parameters while f is fixed. More sophisticated negotiation algorithms can still negotiate f. This is an important achievement since the model should ideally not be more restrictive than the targeted technology.

Definition 11.2. A function f is a *parameterized evaluation function* if it projects the parameter set $p \in P$ and the sequences $[m_1], \cdots, [m_n]$ of measured values onto $\mathbb{B}$.

$f : P \times [M_1] \times \cdots \times [M_n] \mapsto \mathbb{B}$

The meaning of $[M_i]$ is the same as in definition 11.1.

The delivered quality as indicated by $[m_i]$ is acceptable with regard to f and p if
$f(p, [m_1], \cdots, [m_n])$ returns *true*.

The definition is illustrated by Figure 11.6. The function f is a box that takes the metered values for the measurands $m_1, \cdots, m_3$ as input. The parameters p_1 and p_2 allow to tune f. Tuning means adjusting the desired QoS. The box will show green if the metered values satisfy

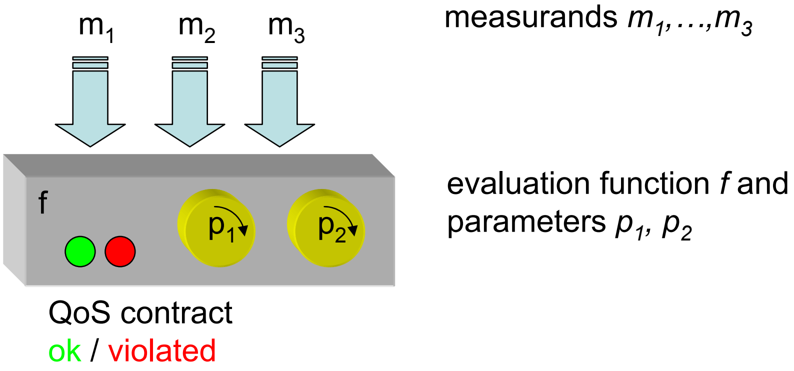


Figure 11.6: Concept of QoS measuring and evaluation

the desired QoS. Otherwise the red light will signal an unsatisfactory QoS. However, the concept of a parameter remains to be defined.

Definition 11.3. f is a parameterized evaluation function as in definition 11.2 with

$$f : P \times [M_1] \times \cdots \times [M_n] \mapsto \mathbb{B}$$

Then the parameter set $p \in P$ is defined as follows.

$$p = (p_1, \cdots, p_k)$$

$p_i \in P_i$, where P_i is the set of all possible values of p_i.

$$P = P_1 \times \cdots \times P_k$$

Each parameter $p_i, 1 \leq i \leq k$ has an attached storage type, a physical unit, and a direction.

$$type_p : \{1, \cdots k\} \mapsto Types$$

$$direction_p : \{1, \cdots k\} \mapsto \{\text{Increasing}, \text{Decreasing}, \text{None}\}$$

$$unit_p : \{1, \cdots k\} \mapsto Units$$

The set *Types* contains all available storage types. *Units* is the set of all relevant physical units.

A parameter may be increasing, decreasing or none of both. If the direction is increasing, a higher parameter value demands a better quality of service. Hence, if $f(p1, [m_1], \cdots, [m_n])$ results in *true* then $f(p2, [m_1], \cdots, [m_n])$ yields *true*, too, if $p2 \leq p1$. The opposite is not demanded. This means, if the delivered quality is acceptable for f tuned with $p1$, it is acceptable for f tuned with a value smaller than $p1$, too. If the direction is decreasing, a lower parameter value demands a better QoS. Thus, if $f(p1, [m_1], \cdots, [m_n])$ results in $true$ then $f(p2, [m_1], \cdots, [m_n])$ yields $true$ if $p2 \geq p1$.

It may happen that parameters are neither increasing nor decreasing. For example f could be defined as follows.

$$f(p, [m_1]) = p^2 \cdot \sum_{i=0}^{l} m_{1_i} < 100$$

$p \in \mathbb{N}$ = parameter
$m_1 \in \mathbb{N}$ = measurand
$[m_1] = (m_{1_0}, \cdots, m_{1_l}) \in [\mathbb{N}]$ = sequence of metered values
l = length of $[m_1]$
$m_{1_i} \in \mathbb{N}, 1 \leq i \leq l$ = metered value

For a certain range $p1 \leq p \leq p2$ the function f yields $true$. Every value smaller than $p1$ or higher than $p2$ will result in f returning $false$. Hence, p has no direction according to the above definition. It is in general not advisable to construct f in such a way that a tuning parameter has no direction. Imagine a home entertainment sound system with a turning knob for the volume and there is no turning direction in which the sound is definitely turned down. This would be tremendously unintuitive. Nevertheless, the meta model allows for parameters without direction, too, since this might be required in some rare cases.

The parameters p of f have some similarities with QoS dimensions of QML. However, these QoS dimensions are more related to the measurands as discussed in subsection 11.2.3. A comparison with QML can be found at the end of this chapter in section 11.5.

Frequency of Evaluation

Figure 11.7 illustrates that it is very important to specify the frequency at which the metered values are evaluated by f.

The figure shows a fictive bandwidth measurement over 24 hours. The bandwidth is good at the beginning, becomes worse and better towards the end. Whether the average response time is acceptable or not depends on the time frame. The average over the first 12 hours yields a bandwidth of 70 MBit/sec. The same holds for the second 12 hours and for the entire 24 hours. However, the average bandwidth between 6am and 6pm is only 58 MBit/sec. If f is to be evaluated every 12 hours, the outcome of the evaluation is random since it depends on the time frame. It is better to demand that f must yield $true$ for every possible time frame of 12 hours. In Figure 11.7 a measurement happens 4 times an hour. Therefore, f should be evaluated at a frequency of $\nu_f = 4/\text{hour}$.

It is important to distinguish between the frequency ν_f and the frequencies associated with measurands - as discussed in the next section. Obviously, it does not make sense to evaluate f more frequently than measuring takes place. However, a certain amount of new metered values may be required before f is evaluated again. Thus, f may be evaluated at a lower frequency than the frequency of the measurements.

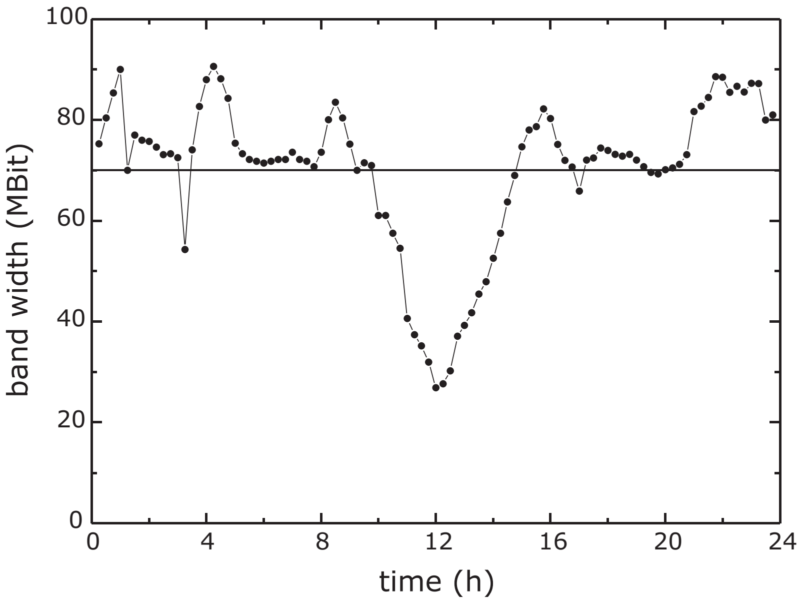


Figure 11.7: Fictive bandwidth measurement

11.2.3 Measurements

To determine the current QoS at runtime the application, middleware, operating system or hardware has to monitor certain measurands m_i. Each measurand is metered at a certain frequency ν_i or upon some event. Hence, a sequence of metered values $[m_i]$ is created for each measurand m_i and the sequence grows by one at the frequency ν_i.

To precisely specify the measurement, the model must express for every measurand m_i:

- *what* to measure,
- *where* to measure, and
- *when* to measure.

Examples for *what* are the current time, the size of a message put on the wire, or the sequence number of the last message received. The *where* determines, for example, whether the time stamp of a message should be determined by the middleware or at the TCP/IP stack. The *when* is either a frequency or an event. For example, the amount of bytes sent could be measured every second, hence, at a frequency of $\nu_i = 1/\text{sec}$. In another example, the size of a message has to be measured whenever a (relevant) message is sent, i.e. when an event occurs.

Specification versus Implementation

In theory, it would perhaps be possible to formally specify the *what*, *where*, and *when* in the PIM. A model transformation tool could automatically generate code that monitors the measurands. However, this is not advisable. The measurands m_i and the evaluation function f *specify* which quality of service is acceptable or not. They do not provide sufficient information to derive an efficient algorithm for the measurement and evaluation of the QoS delivered by a service. This claim is proven by the following example.

To measure the bandwidth, the amount of bytes transferred during the interval of one second is measured at a frequency of $\nu = 1/\text{sec}$. This results in a sequence $m_1 = m_{1_1}, \cdots, m_{1_t}$ after t seconds. The parameter p denotes the minimum bandwidth that is acceptable. The evaluation function f is defined as

$$f(p, [m_1]) = \frac{1}{t} \cdot \sum_{i=1}^{t} m_{1_i} \geq p.$$

t	=	time interval	sec
p	=	parameter	MByte/sec
$m_{1_i}, 1 \leq i \leq t$	=	metered value	MByte/sec
$[m_1] = (m_{1_1}, \cdots, m_{1_t})$	=	sequence of metered values	

Implementing f this way is very inefficient. Instead, f could be updated every second as follows:

$$\begin{aligned} f(p, [m_1]) &= h(p, [m_1]) \geq p \\ h(p, [m_1]) &= \begin{cases} m_{1_1} & \text{if } [m_1] = (m_{1_1}) \\ \frac{(h(p,(m_{1_1}, \cdots, m_{1_t})) * t + m_{1_{t+1}}}{t+1} & \text{if } [m_1] = (m_{1_1}, \cdots, m_{1_{t+1}}) \end{cases} \end{aligned}$$

resulting in a runtime of $O(n)$ instead of $O(n^2)$ after n seconds. In other settings, the hardware might offer a byte counter. Clever QoS monitoring code would read this value - called c here - every second. f would then be defined as

$$f(p, c, t) = \frac{c \cdot 10^{-6}}{t} \geq p$$

c	=	byte counter	#Bytes
t	=	time interval	sec
p	=	parameter	MByte/sec

In this case the implementation of f is entirely different. The measurand m_i (bytes sent during the last second) is never measured directly. Instead, the bytes sent since the QoS contract has been established are measured.

An application that adheres to a certain QoS contract does not necessarily measure the specified values m_i or treat f as specified in the model. However, the application must behave *as if it would* monitor the measurand m_i and apply the evaluation function f. This means, that reasonable monitoring code can hardly be generated by a tool even if the *what*, *where*, and *when* are formally described.

As a consequence of this observation, the meta model uses strings to specify the *what*, *where*, and *when*. The advantage of this approach is that developers can specify the measurement in natural language, which is usually easier than formalization. The drawback is that the missing formalization does not allow tools to make use of the information since computers do not (yet) understand natural language. This drawback is acceptable. The above argumentation showed that there are other inherent difficulties in automatically deriving an implementation from the measurement specification.

11.2.4 Example

This section provides an example of the QoS category response time and how this QoS category can be defined in terms of measurands m_i, the evaluation function f, and parameters p_i. The server guarantees to answer every message in a given time. Otherwise the server is too slow and the contract is broken. Two measurands are required for the detection of time-outs. Measurand m_1 monitors outgoing messages, their timestamp, and message IDs. Measurand m_2 monitors reply messages received by the server. Tables 11.1 and 11.2 specify these measurands.

Name	m_1
Description	Time stamp and ID of messages sent by the client
When	Client sent a message successfully
Where	Middleware
What	(Timestamp of the message, Message ID)
M_1	$Time \times ID$

Table 11.1: Measurand m_1

Name	m_2
Description	Time stamp and ID of messages received by the client
When	Client receives a message
Where	Middleware
What	(Timestamp of the message, Message ID)
M_1	$Time \times ID$
Constraint	The message ID of the received message matches that of the corresponding send message (see m_1)

Table 11.2: Measurand m_2

A slow middleware influences the timeout, because the measurands produce a timestamp when a message enters or leaves the middleware. The time the middleware spends on marshalling and dispatching is added to the total time required to send and answer the message. A different definition of the measurands could ignore the overhead induced by the middleware. In this case the timestamps are created when a message enters or leaves the transport layer of the operating system. The drawback of this approach is that the timeout is not guaranteed on the application level since the middleware can spend any time on marshalling and dispatching without violating the contract. In both cases, the evaluation function requires a parameter p_1 that defines the timeout.

The evaluation function f works on the parameter p_1 and the two lists of metered values $[m_1]$ and $[m_2]$.

Name	p_1
Description	Timeout
Direction	Decreasing
Unit	msec
Type	integer

Table 11.3: Parameter p_1

$$f(p_1, [m_1], [m_2]) = \begin{cases} (\forall m_{1_i} : h(m_{1_i}, [m_2], p_1) = true) & true \\ \text{else} & false \end{cases}$$

The helper function h returns *true* if the message has a corresponding reply message in $[m_2]$ or if the message has been sent during the last p_1 milliseconds.

$$h(x, [y], p_1) \;=\; now - t(x) < p_1 \text{ or } \exists y_i \in y \;:\; id(x) = id(y_i) \text{ and } t(x) - t(y_i) < p_1$$

$time(x)$ = Time stamp of the message x
$id(x)$ = ID of the message x
now = current time

11.2.5 Custom Functions

A contract type can have a set of custom functions in the PIM. These functions can be divided into two categories. The first category consists of query functions, i.e. they are free of side effects. Thus, they do not affect the behavior of the contract. The second category changes the contract.

A good example for the first category is a `getUser` and `getRoles` function of an authentication contract. The contract ensures that only authenticated users can use a component. Business logic might be interested in retrieving the name and the roles of an authenticated user. The contract offers this functionality via the `getUser` and `getRoles` methods. The contract is part of the PIM. Therefore, the `getUser` and `getRoles` functions do not necessarily exist in the PSM, too. For instance, they could be mapped to some functions of the ASP.NET framework. The sole purpose of these functions is to support the modeling of the business logic in the PIM.

The second category of custom functions affects the QoS. The functions `suspend` and `resume` of a contract dealing with timeliness are an example (see Figure 11.22). This contract sums up the computing time required for a certain task (see section 11.4). If the designer wants to model an exemption, i.e. some method calls that are not covered by the contract, he can suspend the contract, perform some method calls outside the contract, and resume the contract afterwards. The `suspend` and `resume` functions affect the QoS.

The distinction between the two kinds of functions is important. Imagine a monitor that wants to track who creates, modifies, and destroys QoS contracts. The monitor must be interested in every invocation of the `suspend` and `resume` functions. In contrast, invocations of the `getUser` and `getRoles` functions are of no relevance to the monitor.

11.2.6 Instances of QoS Contracts

QoS contract types are attached to ports to indicate that these ports support certain guarantees. However, a QoS contract type does not specify concrete values. It only tells that timeliness, availability, bandwidth, etc. guarantees can be made. Sometime this is enough information for the PIM. Concrete values are determined at runtime via negotiation between the components or they are configured during deployment. If the designer wants to specify concrete values in the PIM, he must model a QoS contract instance. A QoS contract type a kind of class and an instance of a class has a value for each attribute of its class. Correspondingly, a QoS contract instance can provide a concrete value for every parameter $(p_1, \cdots, p_n) = p$ as specified in its QoS contract type.

A QoS contract instance *may* have a concrete value for every parameter. It does not have to. For example, the designer could specify a component that has a negotiable uptime, but this uptime is always measured over periods of 10 days. In this case, the QoS contract instance would have a value of “10 days” for one parameter but no value for the other one. If a parameter has no assigned value in the PIM, it is subject to negotiation at runtime or configuration during deployment.

A QoS contract instance does not instantiate the measurands of its contract type. From the view point of the meta model this would be possible. This could be interesting to store series of measurements and depict them in a graph as in Figure 11.7. To make this a really useful feature, it would be necessary to couple the model with testing and debugging tools. This idea is further discussed in the outlook chapter since it is beyond the scope of this thesis.

11.2.7 Notation

QoS contract types in the PIM are depicted as a convoluted box as shown in Figure 11.8. This distinguishes them from classes. The name of the contract type is centered at the top of the box. The box contains three compartments.

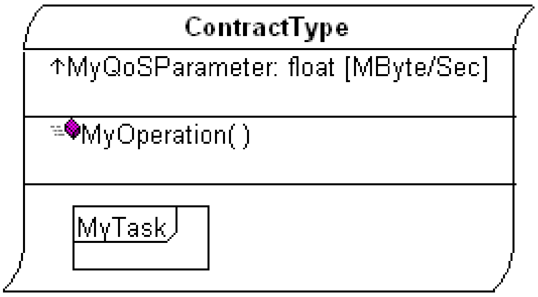


Figure 11.8: Compartments of a contract type

The first compartment contains the parameters. The second one hosts the optional custom functions. The third compartment contains the task, i.e. a sequence diagram. Usually, the task is only shown minimized inside the contract type. The complete task is displayed in an additional diagram. The presentation of the parameters adheres to the following syntax.

$$\begin{array}{rcl} parameter & := & direction\, name\ \ '\!:\!'\ \ type\ \ '\,['\ unit\ ']' \\ direction & := & \uparrow \mid\ \downarrow \mid \text{ none} \\ name & := & [A-Z\ a-z\ 0-9\ \_\ ]+ \\ type & := & [A-Z\ a-z\ 0-9\ \_\ ]+ \\ unit & := & [A-Z\ a-z\ 0-9\ \_\ \%\ /\ ]+ \end{array}$$

An example is

$$\downarrow time : int\,[msec]$$

This defines a parameter named *time*. Its storage type is an integer and its unit is milliseconds. The arrow indicates that smaller values lead to better QoS. The presentation of custom functions follows exactly the standard UML notation for operations. An example is

$$+customOperation(\text{ in } param\ :\ int\ )\ :\ int$$

This specifies a custom function named *customOperation* with one input parameter named *param* of type *int*. The function returns an *int*. The visibility of custom functions is always public since the sole purpose of custom functions is to model the interaction between contracts and components. Thus, the function is prefixed with a +. Alternatively, modeling tools can use an icon.

If the custom function does not modify the contract, i.e. if it does not have side effects (see subsection 11.2.5) then the UML notation for side effect free operations is used, for example

$$+customOperation(\text{ in } param\ :\ int\ )\ :\ int\ \{\text{isQuery}\}$$

Contract types are attached to ports using a realization dependency. A realization dependency is a standard UML element. It is denoted as a dashed line with a hollow arrow head. The arrow points from the port to the contract type. An example is shown in Figure 11.9.

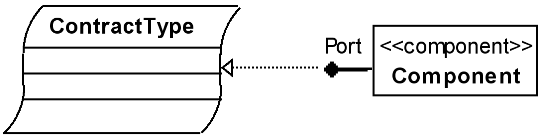


Figure 11.9: Attaching contracts to components

A realization dependency connecting port and contract type can optionally have a name. Furthermore, a realization dependency can have a set of stereotypes attached to it. A component can handle a QoS category in three different modes.

- proactive
- reactive
- monitoring

One of these three keywords can be shown next to the realization dependency. If it is omitted, proactive is assumed. The notation is shown in Figure 11.10

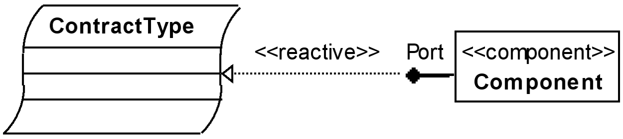


Figure 11.10: Stereotyped contract binding

An additional stereotype is ≪mandatory≫. If a contract is bound to a port using the ≪mandatory≫ stereotype, the component will not operate the port unless this contract is established. For example, a security related contract may be mandatory. If the environment of a component cannot provide the required security level, the component will refuse to operate the port. Finally, the stereotype ≪static≫ can be attached to the realization dependency. This specifies that the contract must be instantiated either in the model – using a QoS contract instance – or during implementation or deployment. Negotiation at runtime is not supported for static contracts. The stereotypes ≪mandatory≫, ≪proactive≫, ≪reactive≫, ≪monitoring≫, and ≪static≫ must not be attached to the contract type, although that may seem natural at first sight. The problem of this approach is that one contract type may be bound to several ports. One port may treat the contract type as mandatory, others do not. Therefore, the stereotypes must be attached to the realization dependency that connects port and contract type.

The evaluation function and the measurands are missing in the diagrams. This is done to foster the clarity of the diagrams. The measurands and the evaluation function are very important for developers who implement a model transformation or a QoS mechanism for a special platform. However, most developers will simply use existing contract types since a new contract type requires work on the model transformation and QoS mechanisms for the target platform. Developers reusing an existing contract type are only interested in those aspects of a contract type that affect them. In fact, the parameters and the custom operations are relevant to them, while the evaluation function and the measurands are just documentation to them. Therefore, the evaluation function and the measurands are only visible in a dialog of the modeling tool. They do themselves not show up in the model. This idea is used in several other places in the UML already. For example, all code pieces attached to methods or actions are usually hidden in the diagrams.

Instances of QoS contract types are depicted as a convoluted box, too. This distinguishes them from UML's notation for objects. Objects in UML are instances of classifiers. Comparably, QoS contract instances are instances of QoS contract types. To distinguish instances from types the name is underlined. Furthermore, a QoS contract instance shows its name and type at the top of the box. The syntax for this is:

$$\begin{aligned} instance &:= name\ ':'\ type \\ name &:= [A-Z\ a-z\ 0-9\ \_\ ]* \end{aligned}$$

An example is:

$$MyPerfContract\ :\ Performance$$

From the definition of *name* follows that a QoS contract instances may be unnamed. An instance has a slot for every parameter of its type. This slot shows the name of the parameter and the associated value. The syntax for slots is as follows.

$$\begin{aligned} slot &:= parameter\_name\ '='\ value\ unit \\ parameter\_name &:= [A-Z\ a-z\ 0-9\ \_\ ]+ \\ value &:= [A-Z\ a-z\ 0-9\ \_\ .\ ]+ \\ unit &:= [A-Z\ a-z\ 0-9\ \_\ \%\ /\ ]+ \end{aligned}$$

An example is:

$$time\ =\ 10\ msec$$

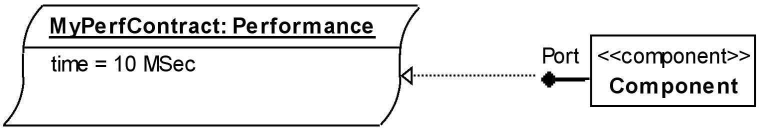


Figure 11.11: Notation of a contract instance

A complete example is shown in Figure 11.11. Contracts can be bound to ports similar to contract types. The semantics are that such a port is able to support this contract. If multiple contracts of the same contract type are bound to the port, the component offers a set of predefined contracts. If the realization is stereotyped as ≪static≫, a selection must be made before runtime. Otherwise contract selection can occur at runtime. The stereotypes ≪mandatory≫, ≪proactive≫, ≪reactive≫, ≪monitoring≫ are supported for dependencies between a port and a contract, too.

11.3 Contract Dependencies

If a PIM statically defines a certain QoS contract instance for every QoS-enabled port, runtime negotiation and adaptation is not needed. In contrast, if the PIM just specifies QoS contract types for some ports, runtime negotiation or deployment configuration is required. In these cases it is important to understand the interdependencies of the different ports and their contracts. One way of addressing this is to model the negotiation and adaptation of QoS contracts in the PIM. Although this is a very interesting research topic, it is out of the scope of this thesis, especially since current QoS-aware middleware platforms have only very limited support for negotiation and adaptation strategies.

A more light weight approach is to make dependencies explicit, but not to describe this dependency in detail. For example, the output quality of a video encoder component depends on its input. This can be modeled with PIQML as shown in Figure 11.12. However, PIQML cannot express the output quality as a function of the input quality. Although that could be achieved with an OCL expression, it is not sufficient in the general case. The output quality depends on additional factors such as memory and processing power constraints. Therefore, PIQML is limited to simpler constructs. A contract dependency can belong to one of the following categories:

- provided/required dependency
- negotiation dependency.

The video encoder is an example for the first kind. The second kind is used to constrain the negotiation. Such a constraint could, for example, be used to model the order in which a component allows to negotiate contracts. The following two sections discuss both kinds of contract dependencies in detail.

11.3.1 Provided/Required Dependencies

The QoS provided by component depends on the QoS that it perceives from its environment. A component can follow one of two possible strategies to cope with the problem. The first strategy is to wait for the requests of clients. If a client demands a certain contract, the component will start negotiating with its environment. The second strategy does the opposite. It negotiates the contract that the component requires from its environment first. The contracts that it can provide to clients are limited by the outcome of this first negotiation. This information is very valuable already in the design phase. If a component follows the second strategy, the developer must specify what the component should demand from its environment. What the component will be able to provide is limited by this decision.

This is quite similar to *planned economy.* For example, the plan in communist states determined a priori how many car wheels are to be produced during one year. The amount of cars that can be sold is limited by the number of produced wheels. The drawback is that such systems do not care how many cars are actually sold. The number of produced car wheels will always stay the same. The benefit of the approach is that it requires – in theory – less organizational overhead, because the system does not need to adapt to a change of demand.

A component following the first strategy will automatically negotiate with its environment to satisfy the demand. This is close to *market economy*. The advantage of market economy is that the system adapts better to the actual demand. Therefore, it is less static than planned economy. The drawback is that the system may spend enormous capacities on negotiation. This may eliminate the advantages of better adaptation.

In the model dependencies between provided and required contracts are represented by UML dependencies. In diagrams they are depicted as dashed lines with open arrow head. The dependency is not drawn between the ports because a port can offer multiple contracts and each contract may have other dependencies. A look at Figure 11.12 unveils that for obvious reasons the dependency cannot be drawn between the contract types, because the diagram shows only one contract type. Instead, the dependency is drawn between the two dashed lines connecting port and contract. If the arrow originates in a provided contract (respectively in the dashed line connecting port and contract) and ends in a required contract, the component uses the planned economy strategy.

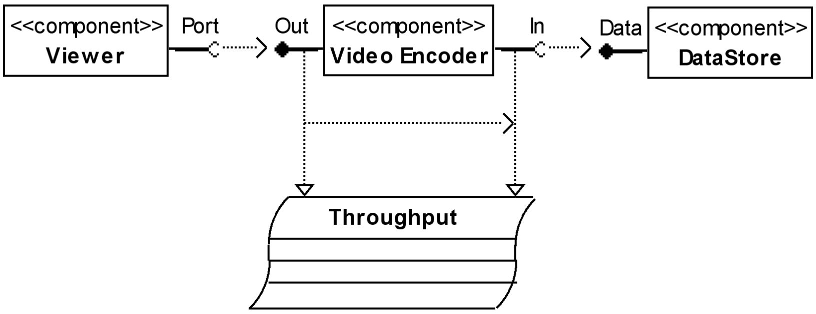


Figure 11.12: Dependency between provided and required contract

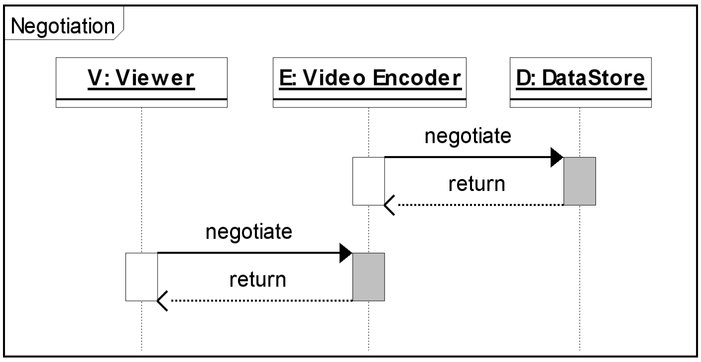


Figure 11.13: Negotiation process for Figure 11.12

When the system shown in Figure 11.12 is started, the video encoder component will first negotiate the contract attached to its required port. Then the viewer component can start negotiating with the encoder about the contract attached to the provided port. This is illustrated by the sequence diagram in figure Figure 11.13. If the arrow points in the opposite direction, the component implements the other strategy – i.e. market economy. This is illustrated by

Figure 11.14.

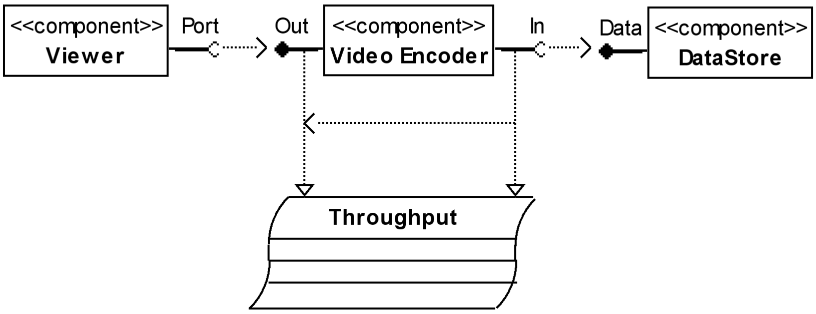


Figure 11.14: Dependency between required and provided contract

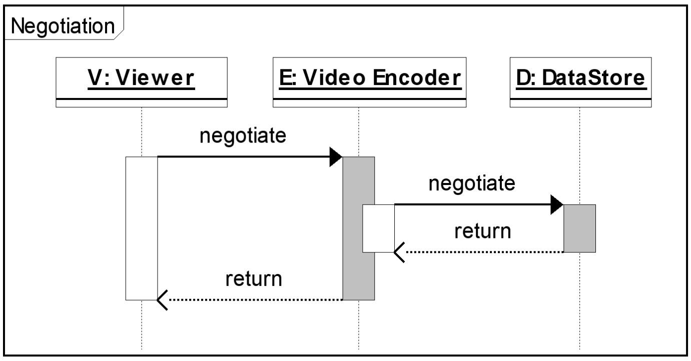


Figure 11.15: Negotiation process for Figure 11.14

When this system is started, the encoder component will not actively negotiate any contract. When the viewer component requests a contract, the encoder will negotiate its required contract with the data store in such a way that the request of the client is satisfied - as far as this is possible. The negotiation process is shown in Figure 11.14.

11.3.2 Negotiation Dependencies

A different kind of dependencies is related to the negotiation process. Imagine a component in the context of banking that analyzes the impact of currency changes for brokers. This component will be used in parallel by many brokers. Speed is of course a consideration since brokers must decide very rapidly. Therefore, the component offers two contracts. One contract is about concurrency. It determines how many users the system can service at a time. The other contract determines how long one analysis will take. Obviously, a system with many users will be slower than one with few users. The negotiation algorithm of the component may be realized in such a way that it requires the number of concurrent users first. Then it can make offers regarding the performance of the computation. The technical reason might be, for example, that the component will be executed on a small server if the number of concurrent users is less than 100. The small server is not very fast, hence, the available performance is quite limited. If the number of concurrent users is greater than 100, the component is executed on a load balanced cluster of fast servers. As a consequence, the component can now offer better performance contracts. Negotiating the contracts in the opposite ordering - hence, performance first, concurrency second - may lead to the unfortunate situation where few users can force the component to be executed on the expensive cluster just because the negotiation demanded a performance of 5 seconds whereas the small server could provide a guaranteed response time of 6 seconds. Restricting the negotiation order prevents this from happening. Dependencies as in the example above are called *negotiation dependencies*.

Negotiation dependencies exist only between provided contracts. They specify in which order a client of the component must negotiate contracts, which contracts are mutual exclusive and which contracts must be negotiated in an atomic manner. This is important in the PIM since two components can be incompatible if they disagree on these topics. Negotiation dependencies can be further subdivided in three classes. The first class restricts the *ordering* of contract negotiations. The second class specifies *mutual exclusions* of contracts. The third class determines the *atomicity* of contract negotiations. That means a component enforces the negotiation about several contracts en block.

Ordering

A component may enforce a certain order on the contract negotiations. This is only important for runtime negotiable contracts. Thus, their connection to the port must not be stereotyped as ≪static≫. UML dependencies are used to model ordering dependencies. For example, the component in the Figure 11.16 insists on negotiating contract `C1` before `C2`.

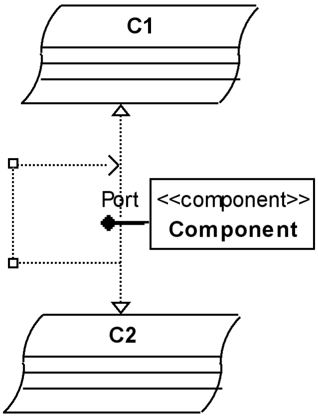


Figure 11.16: Negotiation ordering

Mutual Exclusion

Some QoS contracts can be mutual exclusive. In this case, they are connected with an $\{xor\}$ constraint. This is a standard UML way of marking two elements as mutual exclusive. Figure 11.17 provides an example.

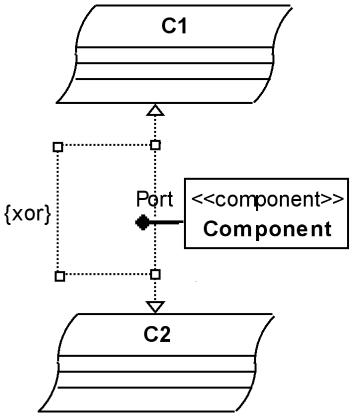


Figure 11.17: Component with mutual exclusive contract realizations

Atomicity

Negotiation strategies can be quite complicated to implement, especially if multiple QoS categories have to be obeyed. A first step towards simplification is to limit the order of QoS contract negotiations. An even simpler approach is to negotiate several QoS contracts en block, hence, in an atomic manner. Atomicity is modeled with a new QoS contract type. The new QoS contract type inherits all QoS contract types that should be negotiated in an atomic manner. A UML generalization dependency is used to model the inheritance. An example is shown in Figure 11.18.

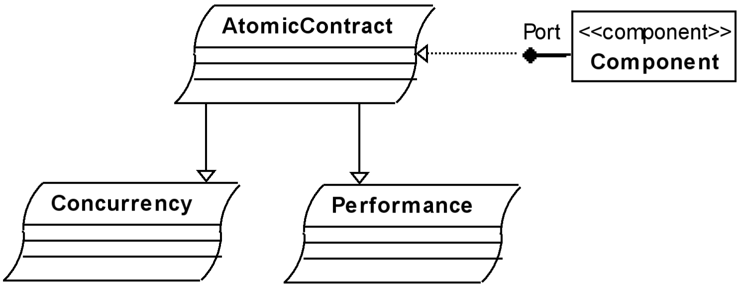


Figure 11.18: Component with atomic contract negotiation

The introduction of a new QoS contract type requires in turn an adaptation of the model transformation. The reason is that the negotiation algorithm is a different one. Atomic negotiation cannot be expressed as a simple concatenation of single negotiations because atomic negotiation has all-or-nothing semantics. If multiple contracts are negotiated atomically, either all contracts must be successfully negotiated or none. This requires – in general – special considerations at the QoS mechanisms. Therefore, the model transformation is affected. The case is different for ordering dependencies. They just constrain the order of negotiations, but they do not imply all-or-nothing semantics.

11.4 Tasks

Some QoS contracts apply equally to the entire communication between two ports. An example is availability. If the component has a failure, it will most likely affect all methods available via the interface of a port. Therefore, an availability contract can be attached to an entire port. The case is different for contracts involving timeliness. Some methods require more computation than others. Therefore, it is appealing to attach different contracts to different methods. A contract about the timeliness of a single method call is only usable in hard real-time systems. The contractually guaranteed time must always be a worst case time since every single method call must complete in the specified time. However, worst case times are often far away from the time that was effectively required. Therefore, the time difference δt between the guaranteed time and the effectively required time for each method invocation may be quite big.

After n subsequent invocations this sums up to $n \cdot \delta t$. This means that the contractually guaranteed time must be much worse than the real performance of the system. This leads to the undesirable observation that the system guarantees only a very long runtime for n calls while its effective worst case runtime is much shorter. Especially enterprise applications are affected by this observation. For an enterprise application it does not matter how long a single method call takes to complete. The real question is: how fast will the entire task complete? The problem is that the contract in the above example is about the worst case time of a *single* method call. It should have been about the worst case time of n invocations instead.

Therefore, it is beneficial to attach QoS contracts to tasks instead of single methods. For example, a task could specify that the client should loop n times and during every loop a certain method is to be called. A timeliness contract for this task would specify a worst case time for the total of n invocations. This shows that the combination of tasks and QoS improves the QoS specification of components.

11.4.1 Accumulative Runtime Analysis

Accumulative runtime analysis is an outcome of research on efficient algorithms. The goal of this approach is to determine worst case times for sequences of operations, hence, tasks. These worst case times are lower than the sum of the worst case times of the single operations. Hence, accumulative runtime analysis delivers sharper worst case approximations for tasks.

A simple example inspired by [Sch99] illustrates the accumulative runtime analysis. Imagine a binary counter with k bits. The counter offers one operation: `increase by one`. The time required for the increment depends on the sequence of 1-bits at the end of the counter.

$$1101\ 0111 + 1 = 1101\ 1000$$

The above increment required the flip of 4 bits. Obviously, the worst case complexity for one increment of a k bit counter is k. Therefore, $n \cdot k$ is a valid worst case approximation for n increments of the counter. This approximation is quite bad.

Some increment operations are cheap since they flip only one bit. Others are very expensive. The idea of the accumulative runtime analysis is to make the cheap operations pay for the expensive ones. An expensive operation flipping i bits from 1 to 0 increases the *potential*. Subsequent increment operations are fast, because few bits have to be flipped from 1 to 0.

$$1101\ 1000 + 1 = 1101\ 1001$$

The above increment operation is cheap since the previous operations flipped many i bits. Therefore, the potential function $\Theta(x)$ is defined as

$$\Theta(x) := [\text{Number of 1-Bits in} x]\,.$$

It is important to notice that

$$\begin{aligned} \Theta(0) \quad &= 0 \\ \Theta(k) \quad &\geq 0. \end{aligned}$$

Let w_i be the real cost, i.e. the number of bit operations, of the i-th increment operation. Then

$$a_i = w_i + \Theta(k) - \Theta(k-1)$$

is the accumulative cost of the i-th increment operation. The i-th operation switches $w_i - 1$ bits from 1 to 0 and one bit from 0 to 1. Thus, the potential Θ is decreased by $(w_i - 1) - 1 = w_i - 2$. It follows that

$$a_i = w_i - (w_i - 2) = 2.$$

Therefore, an approximation for n subsequent increments starting by 0 yields

$$\sum_{i=1}^{n} a_i = 2n$$

which is much better than the kn determined above. It remains to be proven that

$$\sum_{i=1}^{n} w_i \ \leq \ \sum_{i=1}^{n} a_i$$

i.e. that the real costs are always smaller than the accumulative costs.

$$\begin{aligned}
\sum_{i=1}^{n} a_i &= \left(\sum_{i=1}^{n} w_i\right) + \sum_{i=1}^{n} (\Theta(i) - \Theta(i-1)) \\
&= \left(\sum_{i=1}^{n} w_i\right) + \Theta(n) - \Theta(0) \\
&= \left(\sum_{i=1}^{n} w_i\right) + \Theta(n) \\
&\geq \sum_{i=1}^{n} w_i
\end{aligned}$$

This proves that the accumulative runtime analysis improves the approximation of the runtime for i subsequent increment operations. Figures 11.19 and 11.20 illustrate how the achievement of this analysis can be used in a model. The analysis has proven that the number of bit operations is linear in the number of increment operations. The number of bits does not influence the runtime.

Based on this theoretical result, the contract of the task depicted in Figure 11.19 can be evaluated. The contract demands 100 iterations. The time for the entire task is 10 msecs. Invoking the `clear` operation requires t_c msecs. Hence, $10-t_c$ msecs remain for 100 invocations of the `increment` function. From

$$\sum_{i=1}^{100} w_i \leq \sum_{i=1}^{100} a_i = 2 \cdot 100$$

follows that 200 bit flips have to be executed for 100 `increment` invocations. Let t_f be the time in msec required for a bit flip. Then the following constraint must always hold if the contract is valid.

$$t_c + 200 \cdot t_f \leq 10\text{msec}$$

Without the accumulative runtime analysis the constraint would have been

$$t_c + 100 \cdot 64 \cdot t_f \leq 10\text{msec}$$

for a 64-bit integer, because the worst case runtime for a call to `increment` would be $64 \cdot t_f$, i.e. 64 bit flips. This is a difference by a factor 64/2 when compared with the results of the accumulative runtime analysis. Hence, using tasks and accumulative runtime analysis allows for significantly sharper worst case guarantees.

Incrementing a bit counter seldom is a problem in practice. Scientists have invented specialized algorithms for real world problems and analyzed them using accumulative runtime analysis. For example, splay trees [ST85] provide better worst case performance for inserting and searching elements in ordered dictionaries. Fibonacci trees speedup Prim's algorithm [Pri57] for minimal spanning trees and Dijkstras algorithm for shortest paths. They have in common that they have a good worst case performance for an entire task. The worst case performance for a single invocation can be comparably bad. Thus, QoS specification languages such as QML that attach contracts to methods instead of tasks cannot take advantage of these algorithms. Accumulative analysis can be used for memory complexity, too. Hence, it can be useful for multiple QoS categories.

11.4.2 Modeling Tasks

The UML provides two diagram types that can be used for modeling tasks.

- State charts
- Sequence diagrams

State machines (finite automatons) can be used to describe the allowed message sequences. UML's state machines are even more powerful than finite automatons. They feature so-called history states. These history states can be treated as some kind of stack. The combination of finite automaton and stack is known as push down automaton, which is able to parse context free grammars while finite automatons accept regular grammars only [HMU00]. Despite the expressive power of UML's state machines, the approach presented in this thesis uses sequence diagrams to express tasks. Sequence diagrams put an emphasis on the time. The messages in a sequence diagram are vertically stacked according to their chronological order. State charts, as already indicated by their name, put the emphasis on the change of state instead. The decision in favor of sequence diagrams and against state charts is merely based on notation issues. Conceptually, state charts would be possible, too. This is underlined by the transformation of sequence diagrams into state machines later in this chapter. State machines are easier to handle in a formal way while sequence diagrams are a better guide to the eye. Thus, a task is depicted as a normal UML 2.0 sequence diagram. The involved parties are depicted as objects. In the example presented in Figure 11.20 the task involves four objects. The first two objects (from left to right) represent the two components that carry out the task. The right most object represents a QoS contract instance. The corresponding component diagram is shown in Figure 11.19.

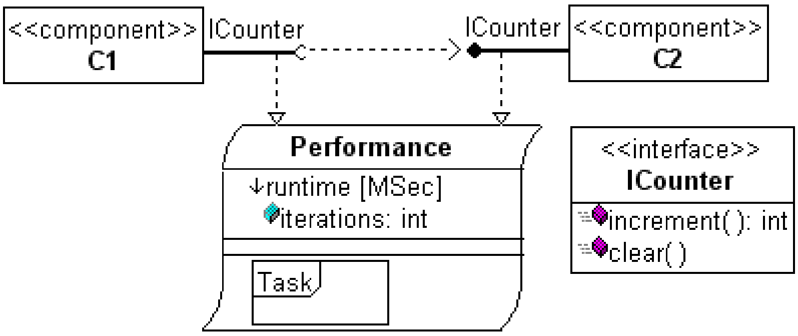


Figure 11.19: Component diagram for a simple counter

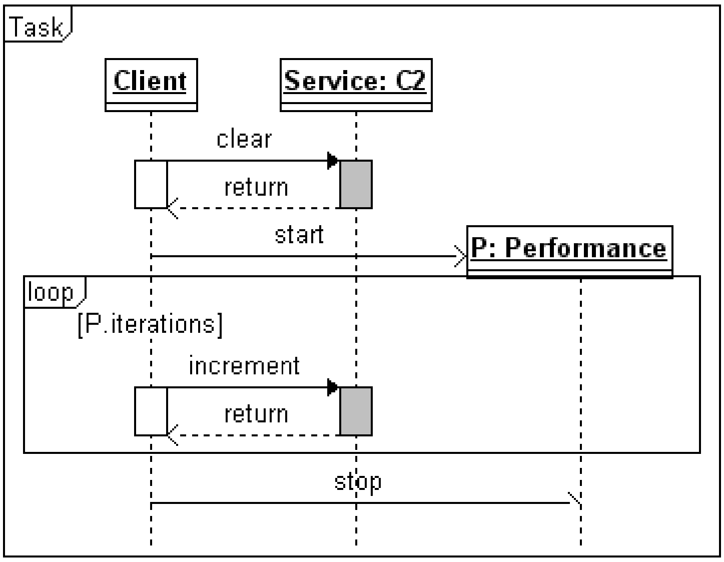


Figure 11.20: Task associated with the `Performance` contract

If a contract is created, all subsequent messages are subject to this contract. Therefore, the example in Figures 11.19 and 11.20 shows that the contract type `Performance` is only applicable to the `increment` method of the `ICounter` interface. The `clear` method is not affected by the contract. The call of the `increment` method is hosted inside a loop. The number of iterations is defined by `P.iterations`, which is a parameter of the `Performance` contract. Thus, the entire task consists of a call to `clear` and several calls to `increment`.

Figure 11.19 shows that the `Performance` contract has in fact two parameters. The first parameter, named `runtime`, is a parameter p of the evaluation function in the sense of definition 11.3. It defines how much computing time the `increment` operations will require in total. The second parameter, named `iterations`, is not directly associated with the evaluation function. Instead, `iterations` is a parameter of the task. Thus, an instance of this contract type will have a concrete value for the number of iterations. The two kinds of parameters are depicted differently in the model to underline the difference. Parameters of the evaluation function have an arrow in front and their physical unit is shown inside brackets. Parameters of the task are depicted like attributes of classes in UML 1.4.

Figure 11.19 does not show any concrete value for the parameters. This means that these parameters are either negotiated at runtime or configured during deployment. Sometimes concrete values can already be determined during design time. Figure 11.21 shows concrete values for each parameter. The components are bound to a task with 100 iterations and the time required for the 100 calls to `increment` must be $\leq$ 10 msec. Thus, the contract does not require negotiation at runtime.

A mixture of both is possible, too. Thus, the contract instances have concrete values for *some* parameters but not for all. Then the remaining parameters will be subject to negotiation at runtime. This is especially useful for task parameters. It is quite unlikely that the count of iterations as specified in Figure 11.19 is precisely known at design time. However, it is more

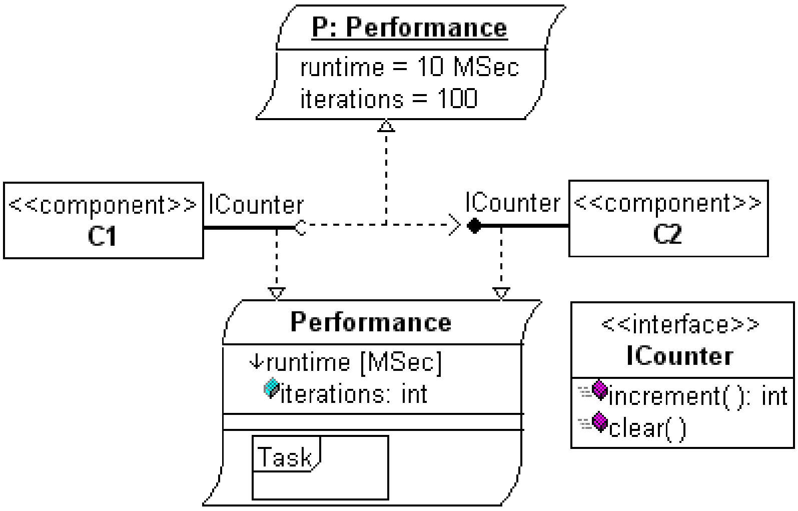


Figure 11.21: Instantiation of the `Performance` contract

likely that the designer knows that the counter may not consume more than 10 msec in total. Therefore, the time can already be specified in the design while the precise number of iterations is determined at runtime.

Some contracts offer an additional set of functions. For example, an authentication contract may offer a function to determine name and roles of the current user. In these cases a component may wish to exchange messages with the contract instance. However, this would be an implementation detail of the component and should not show up in the diagram. All communication between a component and a QoS contract instance that is shown inside the task diagram must have an impact on the QoS of subsequent messages. For example, a QoS contract type may offer methods `suspend` and `resume`. Their usage must appear in the task since they affect the QoS. This is illustrated by Figure 11.22. The client `O1` can call the `read` function as often as it wants to. The time required for the `read` calls is not relevant to the performance contract, because the contract is suspended before and resumed after the `read` calls. It would not be possible to model this task and its QoS properties without the `suspend` and `resume` functions.

11.4.3 Validity Check of Message Sequences

Tasks are used to model the set of allowed message sequences. During runtime the task model can be used for two purposes:

- detecting erroneous message sequences,
- activation/deactivation of QoS contracts.

During runtime the client sends messages to the server side and the server side sends response messages. The task specifies the set V of message sequences that comply with the task. This set may be of infinite size. In order to treat task models in a more formal way, they are projected

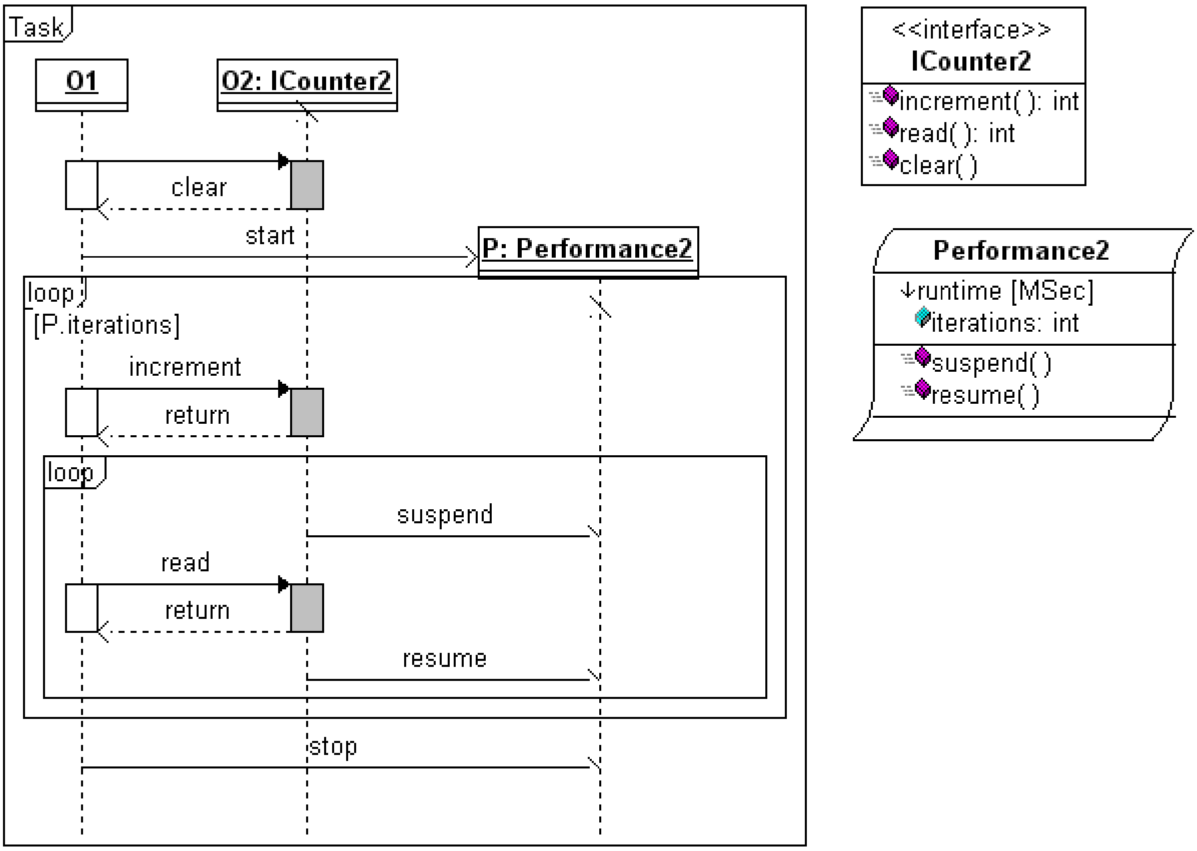


Figure 11.22: The extended `Performance` contract

on formal languages. The set V is treated as the vocabulary of a formal language. Hence, every word $w \in V$ is a sequence of messages. Each word w is composed of single characters, hence,

$$w = (w_1 \cdots w_n) \quad \text{with } w_i \in \Sigma.$$

The set of characters Σ contains at least one element for each method of each interface involved in the task. If a method is not one-way – i.e. if a return message will be sent – a second element is added to Σ. Additionally, for each contract c used in the task the two messages c^a, $c^d \in \Sigma$ are added.

- c^a for contract *a*ctivation,
- c^d contract *d*eactivation.

Client and server monitor the sequence of messages and the activation/deactivation of contracts. These observed messages must yield a word w. If $w \in V$, client and server conform to the task. This conformance covers the correct ordering of messages as well as the correct activation and suspension of QoS contracts. The implications of QoS contracts for the formal language are further discussed in subsection 11.4.4.

In order to check the conformance of message sequences at runtime, an automaton is required that accepts the language V. In many cases a finite automaton - hence, a regular language - is sufficient.

Proposition 11.1. Tasks that do not contain concurrent compartments (marked with `par` in the diagram) and that do not use recursion can be checked by a finite automaton.

Proof. The proof shows that the task model can be reduced to a regular expression. Regular expressions have the same expressive power as finite automatons [HMU00].

The input alphabet of the regular expression is Σ.

One message can be represented by a character in Σ. Hence, a single message corresponds to a valid regular expression.

The `opt` compartment can be reduced to $(r?)$ where r is the regular expression inside the `opt` compartment.

The `alt` compartment can be reduced to $(r_1|r_2|\cdots|r_n)$ where r_i is the regular expression of the i-th alternative in the `alt` compartment.

The `loop` compartment can be reduced to $(r*)$ where r is the regular expression inside the `loop` compartment.

The proof follows from the fact that two concatenated regular expressions are in turn a regular expression. □

If recursion is allowed, a push down automaton - hence, a context free language - is required.

Proposition 11.2. Tasks that do not contain parallel compartments can be checked by a push down automaton.

Proof. A finite automaton is created from the task model. The transition that enters a recursion puts a symbol on the automaton's stack to indicate where to continue when the recursion returns. The final state of the recursively called automaton will check for these symbols on the stack. If there is one, it is popped from the stack and the automaton transits to the state where recursion started. □

Each finite automaton can be reduced to a minimal automaton. Such minimal automatons have the property that they do not contain dead-end states.

Observation 11.1. The automatons constructed from the task model do not contain dead-end states. Thus, if the automaton already read the word x then $\exists\ y \in \Sigma^*$ such that $xy \in V$.

Parallelism

Tasks can contain `par` compartments, which host sub-tasks that can be executed in parallel. In such cases the first sub-task is treated as if it is executed by the thread that entered the `par` compartment. All other sub-tasks are executed by threads that are forked from the one that entered the `par` compartment.

A consequence of this treatment is that a main-thread executes during the complete task. This main-thread executes always the first sub-task in every `par` compartment. For this main-thread it is possible to construct a word $w \in V \subseteq \Sigma^*$ representing the message sequence. For each additional sub-task a new thread is started. The new threads will be subject to another automaton. Their automaton is constructed from their respective sub-task.

Thus, a total of $1 + s - p$ automatons is required if p `par` compartments contain altogether s subtasks. One automaton is required for the main-thread. Each sub-task except for the first one requires an additional automaton, hence, $s - p$ additional ones.

This shows how to check the conformance of message sequences with automatons in the presence of parallelism. However, it has to be noted that the monitoring entity (usually some kind of middleware) must be able to associate each message with a logical thread. This can be implemented by sending some context information along with every message. This technique can be applied at least to CORBA, .NET Remoting, and SOAP based services. This leads to the following observation.

Observation 11.2. The compliance of message sequences to a given task can be checked with a set of push down automatons if the compliance checking entity can associate every message with a logical thread.

11.4.4 Validity Checks of QoS

The constructed automatons can be deterministic or non-deterministic. According to [HMU00] their expressive power is the same. However, even a deterministic automaton does not necessarily have the property that it can non-ambiguously associate every real message with a message of the task model. This is illustrated by the following example.

The task shown in Figure 11.23 can be checked with a simple finite automaton. The corresponding regular expression is ab. The automaton is shown in Figure 11.24.

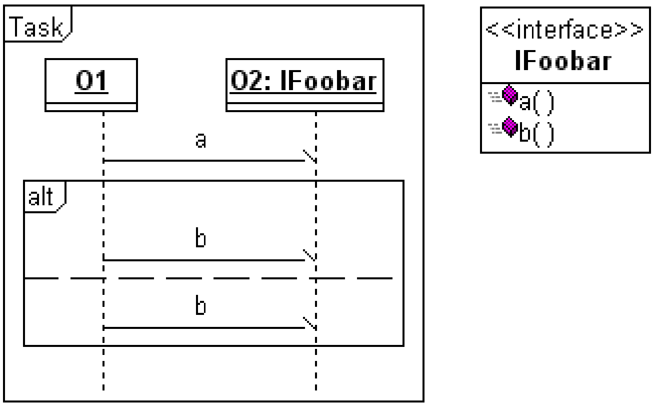


Figure 11.23: Ambiguous task

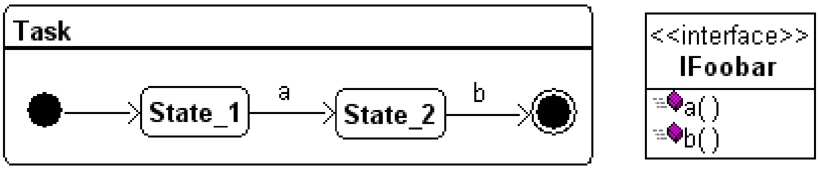


Figure 11.24: Automaton for Figure 11.23

Obviously, the automaton does not care whether the message b belongs to the first or second sub-task in the `alt` compartment. This is irrelevant if no QoS contracts are involved. However, if the sub-tasks in the `alt` compartment are subject to different QoS contracts, it makes a significant difference to which sub-task a message belongs. Such a case is shown in Figure 11.25.

The task in Figure 11.25 can be checked with the regular expression

$$a(c^a b c^d | b).$$

Hence, the regular expression accepts only the words ab and $ac^a bc^d$. The automaton can check whether the QoS contract has been activated and deactivated in accordance to the task model. However, it is desirable, too, that the automaton could automatically turn on or off the required QoS contracts. In this case the client would just exchange messages, a and b in the above example, with the server. The automatons on client and server side will automatically establish the required QoS contracts in accordance with the task. Thus, in the above example, the messages c^a and c^d would be automatically inserted by the automaton.

In the above example this cannot work, because the client would always send the message sequence ab. The automaton cannot detect whether it was the intention of the client to use the QoS contract c or not. This leads to the following observation.

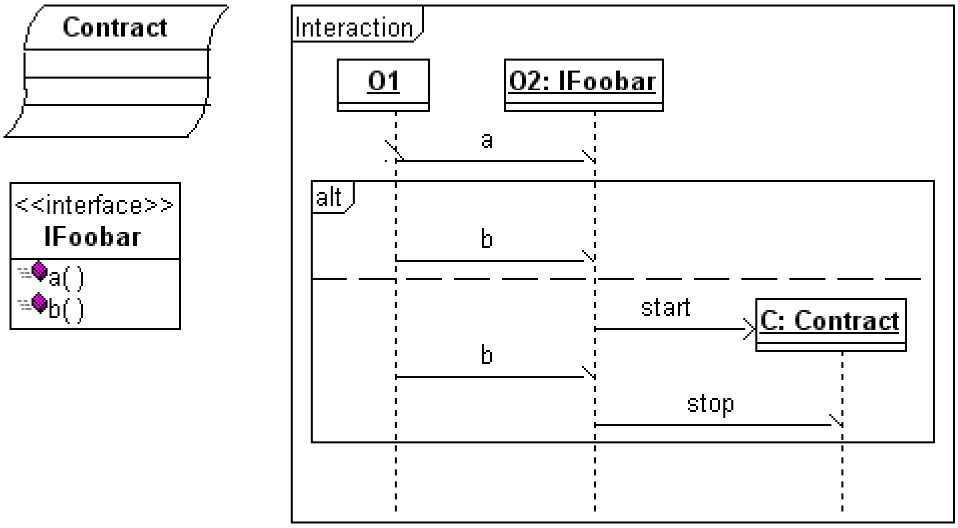


Figure 11.25: Ambiguous task with a contract

Observation 11.3. An automaton constructed from a task model is in general not able to automatically turn on or off the QoS contracts. It can just check whether they have been turned on or off in compliance with the task model.

11.4.5 Automatic QoS Contract Activation

Observation 11.3 only covers the general case. For a large subset of tasks it is possible to automatically activate the required QoS contracts. Hence, Σ is partitioned into Σ_I and Σ_C. The set Σ_I contains the symbols that correspond with methods of the interface. The set Σ_C contains all messages associated with each contract c, i.e. the messages c^a and c^d.

The client is supposed to send and receive only messages that correspond to a symbol in Σ_I. The other messages are automatically sent or processed by the automaton. Such an automaton must be able to determine the QoS contracts that are applicable to each message. That means, if the automaton already saw the message sequence $w = (w_1 \cdots w_{n-1}) \in \Sigma_I^*$ and it reads message $w_n \in \Sigma_I$ then the QoS applicable to w_n can be determined. More formally spoken, let

$$M = (Q, \Sigma, \delta, q_0, q_f)$$

Q = set of states
Σ = input alphabet
$\delta : Q \times \Sigma \mapsto P(Q)$ = transition function
$P(Q)$ = the power set of Q
$q_0 \in Q$ = start state
$q_f \in Q$ = final state

be the non-deterministic automaton that accepts the language V over the input alphabet

$$\Sigma = \Sigma_I \cup \Sigma_C.$$

Furthermore, let

$$R(a) : \Sigma^* \mapsto \Sigma_I^*$$

be a function that strips all characters from a that belong to Σ_C.

Proposition 11.3. A task model can feature automatic contract activation if its automaton

$$M = (Q, \Sigma, \delta, q_0, q_f)$$

holds the property

$$\delta(q_0, w_1) \neq \{\} \ \wedge \ \delta(q_0, w_2) \neq \{\} \Longrightarrow w_1 = w_2 \ \vee \ R(w_1) \neq R(w_2)$$

$w_1, w_2 \in \Sigma^*$ = arbitrary words over Σ.

Proof. Let $a = (a_1 \cdots a_{n-1})$ be the total sequence of messages sent and received and $a_n \in \Sigma_I$ the next message sent or received by the client (or server respectively).

If $\delta(q_0, a) \neq \{\}$ then a is a prefix of some word $w \in V$ because of observation 11.1.

There exists only one word $c \in \Sigma_C^*$ such that aca_n is a valid prefix of some word $x \in V$.

Assuming the opposite implies that $c_1, c_2 \in \Sigma_C^*$, $c_1 \neq c2$ and $y \in \Sigma^*$ exist such that $w_1 = ac_1a_ny$ and $w_2 = ac_2a_ny$ would both be in V. In this case w_1 and w_2 differ only in characters of Σ_C. Hence, $R(w_1) = R(w_2)$, which contradicts the assumption.

Thus, $c \in \Sigma_C^*$ is uniquely defined and the QoS contracts applicable to a_n can be determined by the automaton. ☐

Not every task model supports automatic contract activation. Furthermore, it depends on the model transformation and the targeted platform whether contract activation will be supported.

11.4.6 Tasks and the Evaluation Function

In contrast to QML (see section 11.5), QoS contracts can be bound to a task, hence to sequences of method calls. If, for example, a time contract is attached to a sequence of calls, the semantics can be two fold.

- The contractually guaranteed time is the total time needed for all method calls together.
- The contractually guaranteed time is the maximum time of each single method call.

Which possibility applies depends on the evaluation function. This is illustrated by the following example. The measurands are the same in both cases. Let m_1 record the timestamp of messages sent from client to server. Let m_2 record the timestamp of the reply message. Hence, $[m_1]$ and $[m_2]$ are sequences of timestamps.

If the contractually guaranteed time is the total time, the evaluation function f must be evaluated when the sequence of calls terminates.

$$f = \left(\sum_{i=0}^{len(m_1)} (m_{2_i} - m_{1_i}) \right) < p$$

$p \in P \quad = \quad$ contractually guaranteed time [msec]

In contrast, if the contractually guaranteed time is a constraint on the time needed for each single method call, f is defined differently.

$$f = (m_{2_i} - m_{1_i}) < p \ , \ \forall\ 1 \leq i \leq len(m_1)$$

$p \in P \quad = \quad$ contractually guaranteed time [msec]

The first case is usually used in the context of enterprise applications, since the time of a single invocation does not matter. The second case is more important for real-time applications. QoS specification languages that do not support tasks are always restricted to the second case.

11.4.7 Tasks versus Synchronization Contracts

Synchronization contracts determine the order and possible parallelism of messages sent between the component and its environment. Therefore, tasks seem to interfere with synchronization contracts since both constrain the allowed sequences of messages. If a component features a synchronization contract and QoS contracts with associated tasks, both constraints apply. Thus, the allowed sequence of messages must conform to the tasks and to the synchronization contract. It is quite likely that the message sequence defined by the synchronization contract and the message sequence defined by the task differ only by messages $\in \Sigma_c$, i.e. messages exchanged with QoS contracts. Hence, if

$$L(T) = L(S)$$

$$\begin{aligned} L(S) &= \text{language over } \Sigma_I \text{ defined by the synchronization contract} \\ L(T) &= \text{language over } \Sigma_I \text{ defined by the task} \end{aligned}$$

then the task is a duplication of the synchronization contract. Note that the language $L(T)$ does not contain messages $\in \Sigma_C$, although they may be part of the task. To overcome this duplication, the task replaces the synchronization contract and is moved outside the QoS contract. Dropping the synchronization contract is no problem if $L(T) = L(S)$. Figure 11.26 illustrates the solution. The task does now belong to the component. In Figures 11.19 and 11.20 the task belongs to the QoS contract instead. The drawback of this approach is that two contracts are mixed: synchronization contracts and QoS contracts.

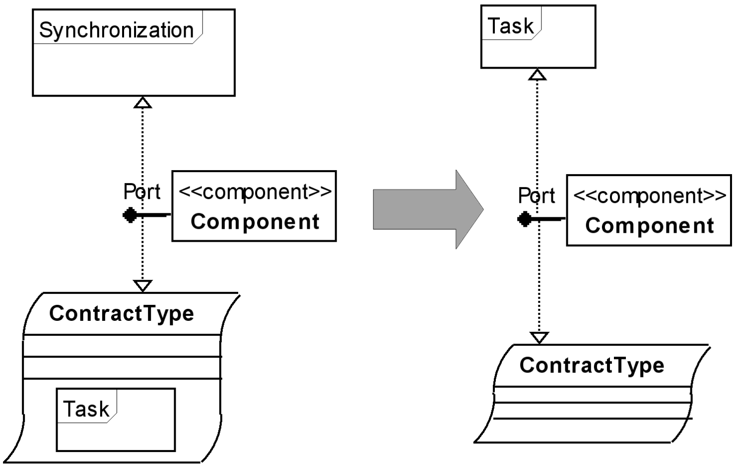


Figure 11.26: Merging task and synchronization contract

However, in some situations QoS contracts must indeed constrain the allowed message sequences more than the synchronization contract, i.e. $L(S) \neq L(T)$. It is desirable that the set of message sequences allowed by the task of a QoS contract is always a subset of the message sequence set allowed by the synchronization contract. As shown in subsection 11.4.3 the set of allowed message sequences can be treated as a formal language over the alphabet

$$\Sigma = \Sigma_I \cup \Sigma_C.$$

Thus, if

$$L(T) \subseteq L(S) \tag{11.1}$$

then all message sequences allowed by the task are also allowed by the synchronization contract. It appears that Equation 11.1 is in general not decidable. Nevertheless, in many constraint settings, the problem is decidable.

Proposition 11.4. If languages $L(S)$ and $L(T)$ are regular then Equation 11.1 is decidable.

Proof. The intersection of two regular languages is again a regular language. Thus,

$$L(M) := L(T) \cap L(S)$$

is a regular language. If

$$L(M) = L(T)$$

then every $x \in L(T)$ must be in $L(S)$, too. Hence, $L(T) \subseteq L(S)$. According to [HMU00] the equivalence of two regular languages is decidable. It follows, that $L(T) \subseteq L(S)$ is decidable, too. □

Unfortunately, this result cannot be applied to more complex languages.

Proposition 11.5. If languages $L(S)$ and $L(T)$ are context free, Equation 11.1 becomes undecidable.

Proof. If the proposition was wrong then

$$L(S) \subseteq L(T)$$

and

$$L(T) \subseteq L(S)$$

would be decidable. If both properties are satisfied, it follows that

$$L(T) = L(S)$$

and the problem is decidable. However, according to [Mar02] the equivalence of two context free languages is undecidable. Hence, the assumption is wrong, and the proposition is proven. □

It is not even possible to check whether $L(T)$ and $L(S)$ are mutual exclusive, since the problem

$$L(T) \cap L(S) = \emptyset$$

is not decidable if $L(S)$ and $L(T)$ are context free languages [Mar02].

11.5 Comparison with QML

Every contract or profile expressed with QML can be expressed with the meta model presented in this thesis. A QML-style QoS contract type can be expressed as a PIQML QoS contract type. A QML-style QoS contract can be treated as a PIQML QoS contract and an unparameterized evaluation function f (see definition 11.1). QML-style profiles can be expressed with tasks. If a QML contract is bound to an entire interface, the contract can be directly attached to the port of a component. If a QML contract is bound to a certain method or a set of methods, the corresponding task creates the QoS contract before this method(s) are invoked and destroys it afterwards.

The main difference between QML's profiles and the approach presented here is that QML binds QoS contracts to interfaces. For the PIQML this is just a special case which is covered by the more general task concept. QML cannot specify QoS contracts that depend on a context, i.e. the current state of the task, in which a method is invoked. Furthermore, QML cannot assign QoS contracts to a sequence of method invocations. For example, it is not possible to specify that the total time required for one call to `foo()` and three calls to `bar()` should not exceed a certain time. Thus, QML has less expressive power than PIQML.

QML does not foster reuse. Each QML contract corresponds to a different *unparameterized* evaluation function. Therefore, each new contract represents a new evaluation function. Different evaluation functions – i.e. different QML contracts – may require new QoS mechanisms, monitoring code, and adaptation strategies. It is an yet unsolved problem to automatically determine whether an existing QoS mechanism is suitable for a certain QML contract. Hence, in general existing solutions cannot easily be reused. Therefore, the evaluation function as used in this thesis can be parameterized. This allows developers to tune the evaluation function while the QoS mechanisms, monitoring code, and adaptation strategies can be reused.

The semantics of QoS dimensions in QML are unclear. QoS dimensions have to be measurands, otherwise it would not make sense to express constraints upon them such as `mean` or `variance`. However, QML does not specify when or how-often they must be metered. Furthermore, it is unclear over which time interval the mean value has to be measured. The importance of such information is discussed in section 11.2.2.

The set of statistical functions offered by QML is limited. QML supports `mean, variance, percentile`, and `frequency`. The functions `mean` and `variance` are self-describing. `Percentile` is used for the specification of statistical distributions on numeric QoS dimensions. `Frequency` is used for the specification of statistical distributions on enumerated QoS dimensions.

Every statistical analysis needs a certain startup time to gain enough data. QML does not provide means to specify how long such a startup time should be. Two contracting parties have three possibilities. First, they assume a startup time of 0. This may have the effect that the contract is immediately terminated after the first value has been metered. Second, they somehow agree on a startup time. This means that the QML contract cannot sufficiently describe the QoS properties of a component. Third, the contracting parties do not agree on a startup time. In this case the party with the shorter startup time may already treat the contract as violated while the other party is still gathering data for the first statistical analysis.

QML cannot mix several QoS dimensions in one constraint. For example, it is not possible to specify that the delay and the throughput should stay in relation. The following constraint cannot be expressed with QML, because it mixes two QoS dimensions in one expression.

$$\text{variance}(c \cdot \frac{delay}{msec} - \frac{throughput}{MByte/sec}) < 0.3$$

$delay$ = QoS dimension [msec]
$throughput$ = QoS dimension [MByte/sec]
c = correction factor

This example is quite reasonable, since an extreme good delay can be useless to streaming applications if the throughput is extremely weak.

Finally, contract negotiation with QML is very complex. A contract is a set of constraints upon the QoS dimensions, hence, a set of mathematical functions. Comparing two QML contracts is therefore quite difficult. Taking into account that QoS contracts can be negotiated at runtime the time complexity of negotiation should not be underestimated. The approach presented in this thesis supports parameterized evaluation functions f. Negotiation is much easier if the negotiating parties agree upon f a priori. Hence, they just have to negotiate the set of parameters p at runtime. This is easier and faster than the negotiation of a set of functions.

Chapter 12

Platform Specific Model

The model transformation projects the platform *independent* model (PIM) onto a platform *specific* model (PSM). A platform consists of a choice of hardware, operating system, middleware, services, programming language, and perhaps additional frameworks such as a GUI toolkit. *Hardware* can be important, because a component's QoS properties are often correlated with the performance and availability of the underlying hardware. The *operating system* (OS) influences the PSM, because it offers OS-specific APIs that may be used in the PSM; in the PIM OS-specific APIs are not allowed. The *middleware* influences the PSM significantly. The PSM for a .NET Remoting based component differs tremendously from a component built on top of CORBA, for example. Especially the approaches to integrate QoS in middleware are very diverse. A component may rely on other *services*, for example databases, a lifecycle service [OMG02a], or a naming service [OMG02a]. Databases vary in the API they offer, in the SQL version they understand, and in the way they handle transactions. The PSM can include fragments of source code. Such source code fragments are attached to bodies of methods in class diagrams or to state transitions in state diagrams. Thus, the selected *programming language* influences the PSM. Finally, a component may be built on an additional *framework*, for example a GUI toolkit [WG00] such as Qt [Tro03] or GTK+ [GTK03]. Integrating the event loop of the middleware with the event loop of the GUI framework can be a tedious task.

Just like the PIM, each PSM is an instance of a meta model. The primary candidate for this meta model is standard UML because UML supports the modeling of a software system's realization. However, as outlined in section 12.5 other meta models are more appropriate in certain settings.

It is worth noting that one meta model can be used for multiple platforms. For example, the UML is not affected by the choice of hardware and operating system. Frameworks and services can be modeled in UML as hierarchies of classes and interfaces. Concepts of the middleware are integrated with the UML by special UML profiles, for example [OMG02e, Gre01]. Finally, some programming languages have special concepts that have no representation in UML, for example .NET's delegates [ECM01a] and meta information [ECM01b]. Such programming language concepts may require an additional UML profile, too.

PIM and PSM do not only differ in the platform independence. The PIM is highly declarative, especially with regards to QoS. The PIM specifies *what* the system should do and which QoS properties is must have. In contrast, the PSM is concerned with the question *how* the system works and how the QoS properties are realized. Therefore, the PSM is more complex and difficult to read when compared with the clearness of the PIM. This process is continued by source code generation and by compiling the source code to CPU specific assembly code.

The development process moves from very high level constructs to more technology specific constructs until it reaches a state that allows a machine to execute the software system.

The PSM is not necessarily a model in the sense of a modeling language. It would be possible to transform the PIM directly into source code and makefiles. Source code can be treated as a textual representation of an abstract syntax tree. This tree can be treated as a model in the sense of definition 5.3. Thus, the same transformation technology can be used to generate source code and makefiles. Section 12.5 gives an example for this approach.

In the case of very large scale applications it may be necessary to address a collection of target platforms. For example, a company may use ASP.NET & IIS (Active Server Pages and Internet Information Server) to communicate with customers via the web while the business logic is implemented in CORBA CCM. This heterogeneous system is modeled homogeneously in the PIM. The model transformation must create a ASP.NET specific model for customer related components and a CORBA CCM specific model for the business logic. These two platform specific models may either be totally disjunctive or they are joined into one model based on a merged meta model.

The following section describes how model transformation can influence the deployment of components by attaching constraints to the PSM. The next four sections demonstrate that the MDA approach used in this thesis can be applied to many quite diverse QoS categories and platforms. Four platforms – ASP.NET & IIS, .NET & COM+ services, DotQoS, and CORBA CCM – are discussed.

12.1 Deployment Diagrams

QoS mechanisms are usually realized in software which is executed on hardware. If this hardware is inappropriate then the desired QoS properties cannot be realized. For example, the best load balancing does not guarantee sufficient performance if the computing machinery is not powerful enough. Another example is availability. The replication of data and services must stay in relation to the availability of the hardware. This implies that the deployment is important for QoS-aware applications.

From the viewpoint of the model transformation the deployment is more difficult to influence. Model transformation can generate design, source code, and state machines. However, the model transformation cannot automatically create a deployment diagram. The PIM does – by intention – not contain any hardware related information. Thus, the transformation does not know a priori on which hardware the components must be deployed.

Therefore, model transformation has two possibilities to influence the deployment. First, the transformation could generate a deployment template. This template shows an ideal configuration of hardware for the software system. Second, the transformation creates constraints that enforce certain properties of the hardware.

These two possibilities are further investigated in the following discussion. The two approaches are not mutual exclusive. A very sophisticated model transformation could produce a deployment template. This template diagram may be modified by the developers to adapt it to their hardware configuration. In a final step, generated constraints check whether the final deployment diagram still satisfies the demanded QoS properties.

12.1.1 Deployment Templates

Deployment templates are especially useful if the software system will be deployed on dedicated hardware. In this case the model transformation could have built-in knowledge of available hardware products. The generated PSM would contain a deployment diagram that shows the hardware required to execute the software system. This includes the computers, their network connection, and may even include information about redundant power supply for mission critical systems. Furthermore, the deployment diagram shows which component to deploy on which machine.

Deployment templates can be very useful for project planning, too. In early phases of the project, designers can model the important components and their QoS properties in the PIM. Then they generate the deployment template to see how expensive the hardware will become. In subsequent steps the QoS properties can be adjusted to reach the optimal relation between cost and capacity.

The drawback of deployment templates is that they do not take existing hardware into account. The PIM does not – and should not – contain hardware specific information. Thus, the model transformation does not have any input regarding existing hardware configuration. Therefore, deployment templates are optimal if new hardware is required anyway.

Modifying deployment templates may result in inadequate hardware configurations. To detect such problems, the model transformation should generate constraints, too.

12.1.2 Examples

In the first example a component is constraint to be deployed on three different machines (nodes in UML terminology). Figure 12.1 shows an adequate deployment of this component.

The constraint is attached to the component. The constraint's boolean expression is not visible in the diagram. It is a Python expression that counts the deployment locations of the component.

```
len( self.deploymentLocation() ) == 3
```

Figure 12.2 shows what happens if the deployment is not acceptable. The constraint raises a warning. This warning is displayed in the dialog and the component in question is displayed in red.

The second example shows the deployment of a database component. In the PIM, the database offered a certain amount of transactions per minute. The database must be deployed on an adequate machine to realize this. Hardware vendors of expensive server machines run benchmark suits to determine the performance of applications such as databases on their machines.

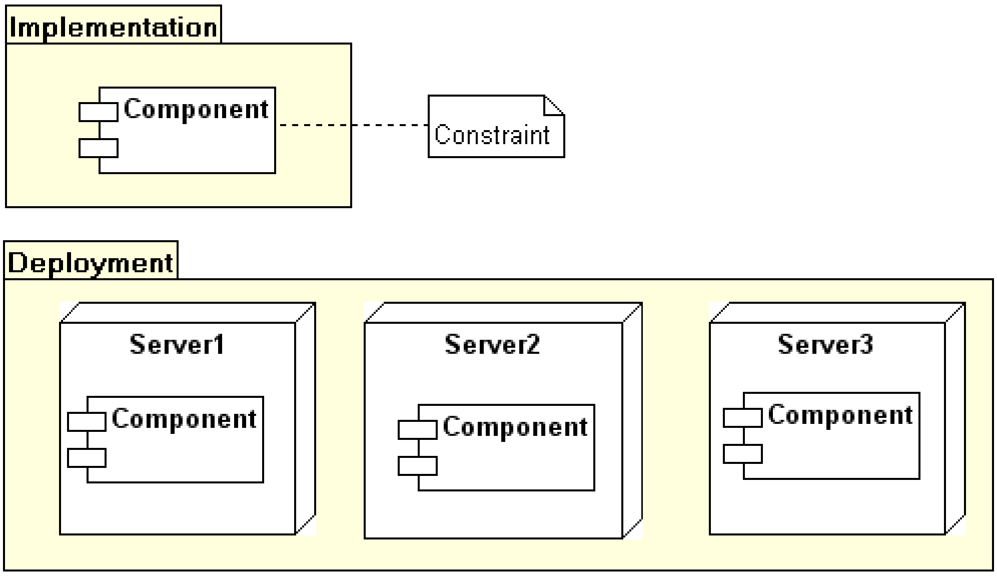


Figure 12.1: Deployment diagram with satisfied constraint

The result of such a benchmark test can be attached to a deployment diagram. For this purpose, tagged values are attached to the UML representation of the machine. A tagged value is a key/value pair. In this example the key is the kind of benchmark and the value is the benchmark's result. Figure 12.3 illustrates this.

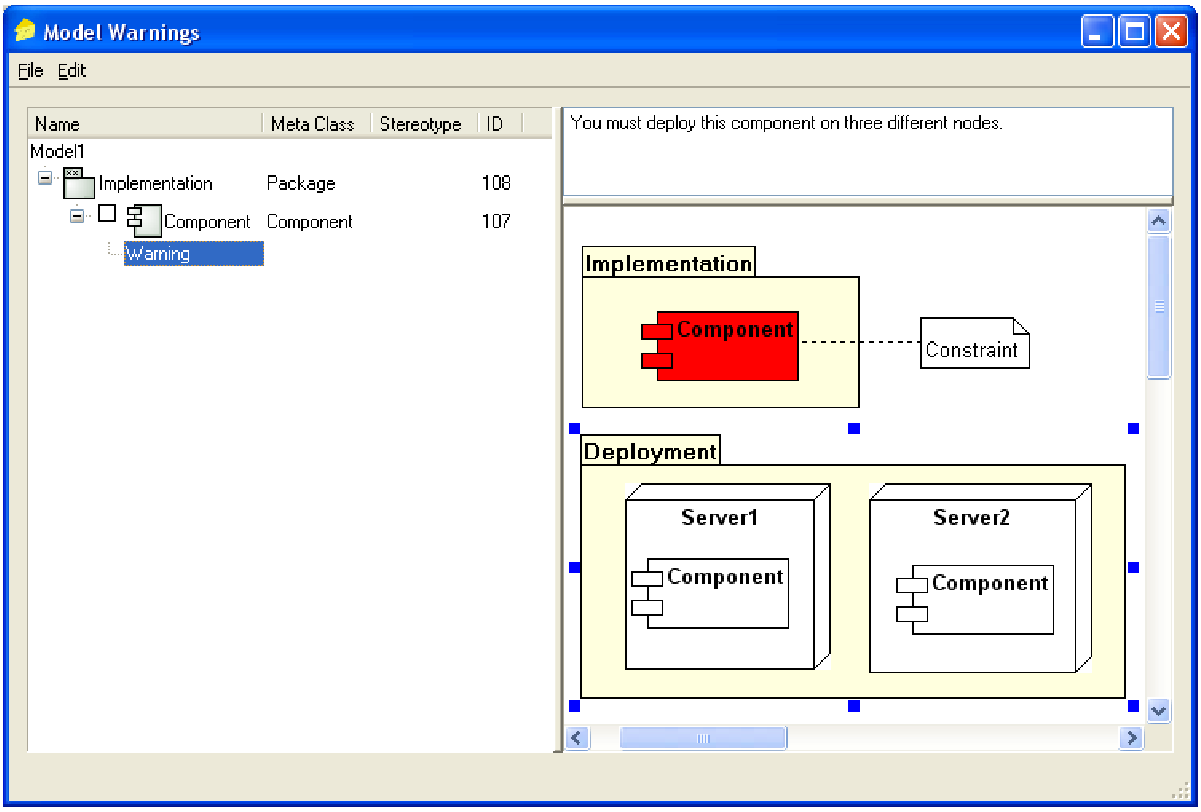
The constraint in this example checks that the node on which the component is deployed has adequate benchmarking results.

```
[t.dataValue() for t in self.deploymentLocation()[0].taggedValue()
               if t.type().name()== "benchmark" ][0] >= 5000
```

The above statement does not check whether the component is deployed at all and whether the tagged value `benchmark` exists. Constraints can become quite large expressions. Therefore, only a short description of the constraint is shown in diagrams.

12.2 COM+

This section shows how a Kafka transformation can be used to transform a PIM into a COM+-specific PSM. COM+ components can be implemented in every .NET language. The following examples use C# [ECM01a]. .NET is object-oriented. Therefore, it is straight forward to utilize an object-oriented modeling language such as UML as a basis for the PSM. Some .NET features have no representation in the UML, for example custom attributes and delegates. Custom attributes can be attached to almost every construct of a .NET compatible programming language. Such constructs are classes, attributes, methods, parameters, enumerations, and so on. Upon runtime a .NET application can use reflection to get hold of these custom attributes. The following code excerpt provides an example:



You must deploy this component on three different nodes.

Figure 12.2: Violated constraint in the deployment diagram

```
[MyClassAttribute( 100, ‘‘Hello World’’ )]
class FooBar {
}
```

Every application programmer can define new custom attributes. In the above example the `MyClassAttribute` has been attached to the `FooBar` class. To add this and other concepts to the UML a profile is required. A UML profile is a set of stereotypes and tagged values [GHS02]. In UML a .NET custom attribute is represented as a stereotyped UML constraint. A constraint can be attached to every UML elements. The same is – almost – the case for custom attributes. Figure 12.4 shows a UML diagram created with Kase which contains a class and a custom attribute.

A COM+ component is a class that is decorated with some COM+-specific custom attributes. These attributes determine which QoS mechanism the component will use: security, transactions, etc. Furthermore, the implementation of this class must obey certain rules which depend on the selected mechanism. The model transformation should generate the COM+ specific parts. The business logic is either derived from the PIM, too, or it has to be filled in by the application developer. The C# source code for a transaction-aware component looks like this:

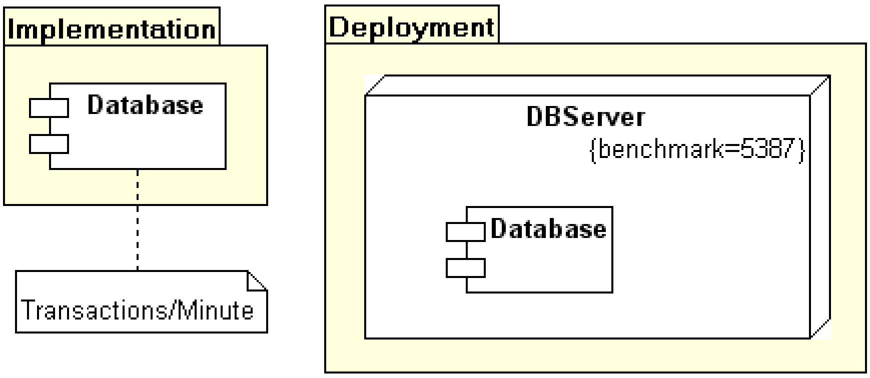


Figure 12.3: Tagged values in a deployment diagram

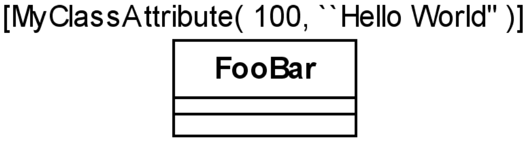


Figure 12.4: .NET custom attributes in UML diagrams

```
using System;
using System.EnterpriseServices;

[assembly:ApplicationActivation(ActivationOption.Server)]
[assembly:ApplicationID(‘‘543534-3455-3245-3234-42342’’)]
[assembly:ApplicationName(‘‘MyApplication’’)]
[assembly:Description(‘‘Some information’’)]

namespace MyNamespace
{
  [Transaction(TransactionOption.Required)]
  public class MyComponent
  {
    public MyComponent() { }
    public void DoSomething()
    {
      if ( !ContextUtil.IsInTransaction() )
        throw new Exception(‘‘No Transaction Context’’);

      //
      // TODO: Allocate resources
      //

      try
      {
        //
        // TODO: Business logic
        //
      }
      catch(Exception e)
      {
        ContextUtil.SetAbort();
        return;
      }
      finally
      {
        //
        // TODO: Release resources
        //
      }
      ContextUtil.SetComplete();
    }
  }
}
```

The custom attribute `Transaction` is used by the .NET runtime. The class `MyComponent` will be put under the control of the COM+ distributed transaction coordinator. Each function that is subject to the transactional behavior starts by checking for a transaction context. If this one is missing, an exception is thrown. Then the method can allocate resources such as a database connection. These resources are later on released in the `finally` clause. The business logic is inside a `try` clause. If the business code fails for some reason then the transaction is aborted. Otherwise the transaction is completed.

The four custom attributes starting with `assembly` tell COM+ to start the component in a separate process. Another option would be to execute the component in the same process as the calling client.

A UML model for the above example is shown in Figure 12.5. The namespace and the class are modeled as UML package and UML class. The custom attributes are displayed on top of the element to which they belong. They are depicted almost like UML constraints. The major difference is that constraints are put inside curly braces. A UML 1.4 component (a physical component) represents the component, hence, a .NET assembly. Therefore, the assembly related custom attributes are attached to the component. The artifact `MyComponent.dll` represents the file in which the assembly will be stored. In UML jargon the class resides on the component and the component's implementation is stored in the `MyComponent.dll` artifact.

The diagram does not show the implementation of the method `DoSomething()`. The dialog shown in Figure 12.6 shows the function's implementation. This source code is generated by the model transformation, too. The generated code is embedded in region constructs:

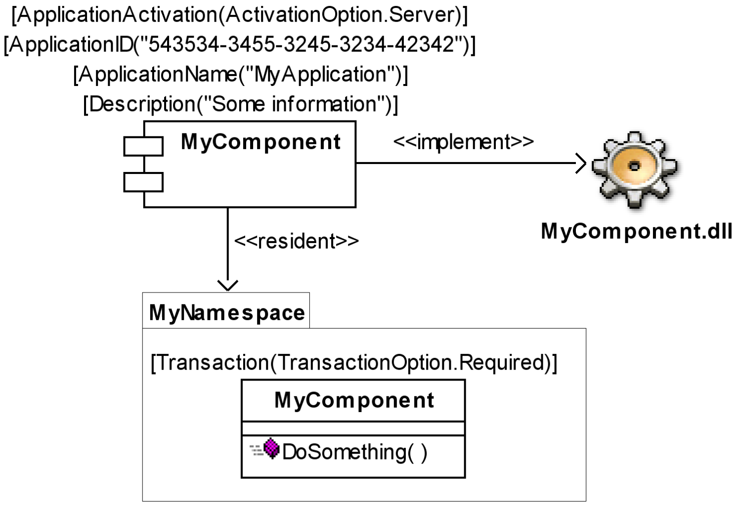

[ApplicationActivation(ActivationOption.Server)]
[ApplicationID("543534-3455-3245-3234-42342")]
[ApplicationName("MyApplication")]
[Description("Some information")]
MyComponent
<<implement>>
MyComponent.dll
<<resident>>
MyNamespace
[Transaction(TransactionOption.Required)]
MyComponent
DoSomething( )

Figure 12.5: Diagram of a transaction-aware COM+ component

```
#region SomeName
...generated code ...
#endregion
```

In VisualStudio.NET the code inside the regions is hidden when the regions are collapsed. This is shown in Figure 12.7. The application developer does not need to know what is inside these regions. He just fills his source code between the regions as indicated by the comments.

The application developer is allowed to modify the code *outside* the regions. The code *inside* the regions is managed by the modeling tools. This is important for subsequent model transformations. The model transformation itself will always produce the same skeleton implementation. A merging algorithm inside Kase's code generation engine automatically updates the code inside the regions or removed/adds regions. For example, if the transaction contract is removed from the PIM then the code regions dealing with transactions will be removed from the source code. Just the business code written by the application programmer would remain. This turns code generation in an easy to use but powerful feature (from the viewpoint of a model transformation developer). The developer of a model transformation just generates a skeleton implementation. Kase's code generation automatically merges the source code written by the application programmer with the source code generated by the model transformation.

Other COM+ features such as object pooling, just-in-time activation, events, and security are handled in a similar way. From the modeling viewpoint they are structurally comparable with the transaction example above. More details can be found in [RAC$^+$02].

12.3 ASP.NET

ASP.NET is a .NET technology. Therefore, the platform specific model is based on UML with some extension. In so far, ASP.NET is similar to COM+. The specialty of ASP.NET is its usage of XML. Some mechanisms such as authentication, authorization, and transactions are configured via XML files. The model transformation must be able to generate the relevant entries in the configuration file.

XML files are – at first sight – text files, but a model is always a graph structure. Hence, the XML structure must be mapped to a graph like structure. A XML file can be represented as a tree structure. The Document Object Model (DOM) [W3C98] is a W3C standard for accessing a XML document programmatically. DOM decomposes the XML text file into a tree structure. This tree structure can be treated as a model. Thus, model transformations can construct XML files. The UML itself does not support this. An extension is required. Mapping the DOM into a UML meta model extension is straight forward. Section B.2 shows the meta model extension for XML.

Figure 12.8 shows a model containing XML tags, a XML file, and a dialog for the editing of XML. Files are represented in UML as instances of the meta class `Artifact`. The content of a file – i.e. an artifact – is not shown in a UML model, although Kase can store the content of the file in the model. XML tags are useful to depict the contents of a XML file or fragments of a file in a model. From the perspective of a modeling language it is not useful to show XML tags in a model. However, XML tags are especially useful in Kafka transformation rules. Using XML tags in Kafka rules, a transformation can add, remove, or modify tags and attributes selectively. The following example will make use of this feature.

The following practical example shows how a Kafka transformation creates a ASPX page that resides in a transaction context. Transactions in ASP.NET pages are enabled by a special tag in the ASPX file. Putting the attribute `Transaction='Required'` in the `Page` tag creates a transaction context whenever the page is processed by the web server.

```
<\%@ Page Transaction="Required"
          language="c#"
          Codebehind="MyPage.aspx.cs"
          Inherits="TransTest.MyPage"
          AutoEventWireup="false"\%>
<html>
   <head>
      <title>ASP.NET Transaction</title>
   </head>
   <body MS_POSITIONING="GridLayout">
      <form id="MyPage" method="post" runat="server">
<asp:button id="btn1" onClick="onButton1" runat="server"
                    Text="Button 1"></asp:button>
<asp:button id="btn2" onClick="onButton2" runat="server"
                    Text="Button 2"></asp:button>
      </form>
   </body>
</html>
```

Figure 12.9 shows the PIM of a simple web application. Kase supports a special PIM meta model for modeling web applications. Each web application is modeled as a component that may feature multiple ports. A port has an attached protocol, HTTP in this example. In the example one port is bound to a transaction contract. The line `mode='Required'` indicates that a new transaction context will be created whenever a page served via this port is requested unless the request happens to be already under control of a transaction context. In this case the page is served in the existing transaction context. The web pages are hosted inside the component. The dashed arrows connecting ports and pages indicate the pages served via each port. Hence, the page `MyPage` will run inside a transaction. A web page can have multiple controls: buttons, text fields, data selectors, and so on. The controls themselves are modeled in the PIM, too. The `Button` control provides an example. Controls can emit signals. The button, for example, emits a clicked signal. Signals can trigger actions. If the `Button1` of `MyPage` is clicked, the transition with the trigger `On_Button1_clicked` will fire and the user is redirected to another web page.

The transformation of a web application PIM into a PSM involves a large set of rules. Only the rules dealing with XML tags are shown here. Previous examples have already demonstrated how Kafka rules can be used to transform PIM constructs into .NET classes, custom attributes, and source code. The transformation of the `MyPage` of the PIM shown in Figure 12.9 into an ASPX page is carried out in three steps. The first Kafka rule matches the web page in the PIM creating the following XML structure and a class named `MyPage`. An ASPX page always consists of a XML file and the corresponding code – called *codebehind* in Microsoft terminology. However, the class and its generation are omitted in the following, since it would not show any new concepts.

```
<\%@ Page language="c#" Codebehind="MyPage.aspx.cs"
          Inherits="MyPage" AutoEventWireup="false"\%>
<html>
   <head>
      <title>ASP.NET Transaction</title>
   </head>
   <body MS_POSITIONING="GridLayout">
      <form id="MyPage" method="post" runat="server">
      </form>
   </body>
</html>
```

The second transformation rule matches all controls and adds the following XML tags.

```
<asp:button id="btn1" onClick="onButton1" runat="server"
                    Text="Button 1"></asp:button>
<asp:button id="btn2" onClick="onButton2" runat="server"
                    Text="Button 2"></asp:button>
```

The third rule checks whether a transaction contract is associated with this page. In this case, the attribute

```
Transaction="Required"
```

is added to the `Page` tag. Figure 12.10 shows the Kafka rules that are used to generate the ASPX file. The `Page` rule creates the ASPX file. The Python expression `@P.name()@` is used to include the name of the PIM page into the filename. The generated ASPX file contains several tags, among them a tag named `form` and one named `Page`. The second transformation rule iterates over all controls hosted by a web page. If the control is a `Button`, an `asp:button` tag is created and added to the `form` tag. The third rule iterates over all component ports that realize a transaction contract. For every page served via a transaction-enabled port, the attribute `Transaction='Required'` is added to the `Page` tag.

12.4 DotQoS

DotQoS is based on Microsoft's .NET technology just like COM+ and ASP.NET. In contrast to these technologies DotQoS is a full featured QoS-enabled middleware. It supports the specification and negotiation of QoS contracts. Furthermore, DotQoS supports the task concept.

The QoS-agnostic transformation for DotQoS works similar to those for the other .NET-based technologies. The QoS-specific transformation is more demanding. QoS contracts must be mapped to .NET classes and custom attributes have to be attached to the generated class. The transformation must select the appropriate QoS mechanisms and add them to the model. In DotQoS, a QoS mechanisms consists of a set of message sinks and optionally a special transport channel as discussed in subsection 10.3.4. Furthermore, custom attributes have to be added to bind the QoS contracts to the implementation classes generated by the QoS-agnostic transformation. Additional source code is required to negotiate the contracts. Finally, the PIM tasks must be mapped to a state machine that is able to determine the QoS contract appropriate for each invoked method. These transformation steps are illustrated by the following example.

Figure 12.11 shows the PIM of a simple e-shop consisting of three components. The `Client` component can order items using the `IShop` interface provided by the `Server` component. The shop in turn uses the `ICard` interface of the `CreditCardSystem` component to handle the payment. Obviously, payment via the Internet needs special precautions regarding security. Hence, a contract is attached to the connector between the client and the shop. Another contract is used between the shop and the credit card company to guarantee a certain performance. It is obvious from the diagram that all messages sent from `Client` to `Server` and vice versa are subject to encryption. The case is more difficult for the QoS category `Timing`. Figure 12.11 shows an interaction diagram titled `MyApp::PaySec::Protocol`. It specifies for which method invocations the time limit applies. Furthermore, the interaction diagram puts a constraint on the possible order of method calls. A client may `query` the shop to get information about available products. Optionally, the client can order items and the shop returns a signed order. The client must either accept the order by signing it with his private key or the client must cancel the order. Malicious clients could attack the shop by invoking `order()` but never `accept()` or `cancel()`. The server would have to store session information for every client and could not release this data until the client accepts or cancels. To overcome this problem the contract includes a timing constraint. Upon reception of an `order()` invocation the server starts the timer. The timer stops when the server replied to either an `accept()` or `cancel()` call. If this transaction fails to finish in time, the middleware has to detect the timeout and cancel the contract. The middleware must provide means for signaling contract violations to the application logic. Since such details are implementation specific, they are not shown in the PIM.

Figure 12.12 illustrates how a subset of the e-shop model in Figure 12.11 is transformed. Some details are omitted for the sake of brevity and the generated source code is of course omitted, too. The two top most packages represent the PIM. The following two packages show the *static view*, which illustrates the realization of the components (left hand) and the QoS mechanism (right hand). The next two packages represent the *physical view*, which shows how classes and other artifacts of the static view are grouped by physical components. These components are then deployed on computing hardware as shown in the *deployment* diagram at the bottom of the figure.

The model transformation maps each PIM component to a package in the PSM. Such a package contains a class for the actual implementation of the functionality - `ClientImpl` and `ServerImpl` in the example. Furthermore, a subclass of `FrameContract` on the server side is generated to handle contract negotiation and monitoring. The contract `PaySec` is mapped to a class. Its attached state machine called `Protocol` implements the invocation protocol (i.e. a task) as specified in the PIM. This transformation can be achieved with Kafka transformation rules, too, as explained in [WUG03a]. The right hand package of the static view shows the implementation of the QoS mechanism. Model transformation can either fetch an existing mechanism out of a repository or create a skeleton implementation as shown here. QoS categories such as `Encryption` are mapped to classes and each QoS dimension to a member variable. The text in brackets on top of the member variables represent .NET attributes, hence, meta-data. The implementation of the QoS mechanism is covered by the two classes `Encryptor` and `Decryptor`. Both inherit a DotQoS-specific interface that allows these classes to be plugged as sinks in the sink-chain.

Models of large scale distributed applications contain many classes, state-machines and other artifacts. In order to deploy such a system a plan is required to determine which artifact belongs where. To ease this undertaking classes, artifacts, etc. are grouped by components - `Server` and `Client` in the example. The same applies to the implementation of the QoS mechanisms. Although the sink-chains titled `ServerChannelChain` and `ClientChannelChain` use a different notation, they represent physical components in the sense of UML1.4, too (see the DotQoS meta model extension in section B.1). Hence, they can be deployed on computing hardware. The ≪local≫ dependency denotes that the component `Client` requires locally (i.e. on the same machine) the `ClientChannelChain` component. The package on the bottom of the diagram in Figure 12.12 shows how those four components are distributed on the computing machinery. A computer called `Console` hosts the `Client` component and the client-side implementation of the QoS mechanism. The `DataCenterServer` is the deployment location of the `Server` component and the server-side part of the QoS mechanism. Using the deployment diagram an automated tool can package the compiled classes and other artifacts to DLLs and copy them to the specified machines.

12.4.1 Multi Category QoS

Multi category QoS requires special precautions when assembling the message sinks to a message path (see Figure 10.4). Each QoS category introduces one or more message sinks. Their ordering is important. For example, one QoS category is security. It injects an encryption/decryption sink into the message path. The second QoS category is bandwidth. It adds compression/decompression sinks to the message path to reduce the required bandwidth. The order in which a message is passed through these sinks is important.

A special DotQoS UML profile [UWG03] allows modeling this in the PSM. Figure 12.13 illustrates this. The diagram features two QoS mechanisms: encryption and compression. Each mechanism is composed of two message sinks. Sinks are depicted as stereotyped classes. The stripes with the convoluted top and bottom edges represent a sink chain. In UML terms they are stereotyped components. Each sink chain is composed of three compartments. Every sink must belong to exactly one compartment. Sinks in the `Head` compartment must always be installed on top of the sink chain. Sinks in the `Arbitrary` compartment can be placed anywhere in the sink chain. Sinks in the `Tail` compartment must be placed at the end of the sink chain. This information tells how the two QoS mechanisms can be installed together in one message path. The resulting sink chain is depicted in Figure 12.14.

The sink chains are UML components. Thus, they can be deployed on nodes in a deployment diagram.

Sometimes two QoS mechanisms cannot be automatically combined as in the above example. For example, two sinks could demand to be on top of the sink chain. In some cases there is no possibility of combining the QoS mechanisms. A new QoS mechanism has to be implemented that covers both QoS contracts. In other case an ordering of the sinks can solve the problem. The model transformation must take care of this. If two QoS contracts are attached to one port and the corresponding QoS mechanisms are known to conflict with respect to the ordering of their sinks, the model transformation must provide a sink chain that solves the problem.

12.5 Qedo

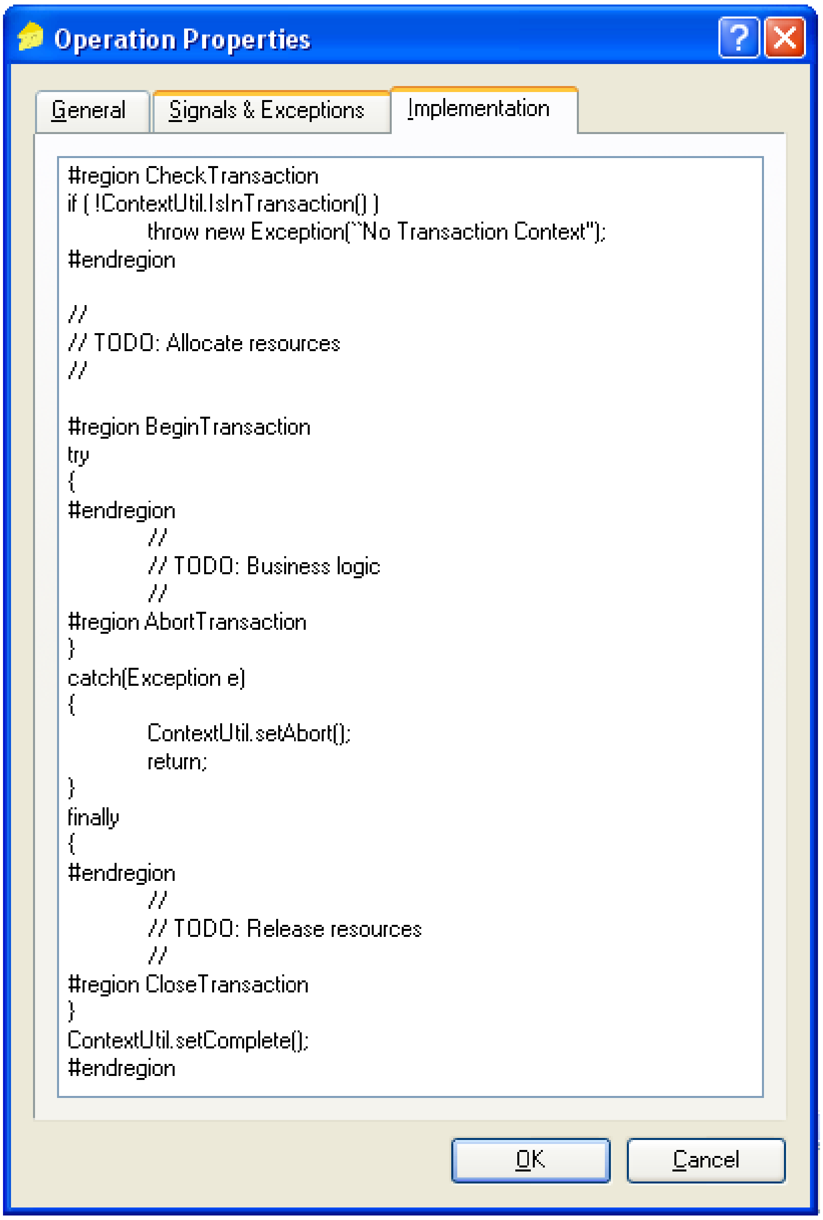
The three PSMs presented so far are based on the UML meta model. This approach is theoretically possible for every object-oriented target platform. However, sometimes a UML profile or even a UML meta model extension might be required. In the case of Qedo [Rit03] (i.e. CORBA CCM [OMG02b]) a PSM based on a UML profile has been proposed by [BR02]. However, CORBA CCM as target platform offers a different option. All CORBA CCM specification languages – as discussed in section 10.3 – are based on a meta model. Thus, model transformation can map PIM concepts directly to concepts of the IDL/CIDL meta model.

In the next step, a code generator scans the model to generate skeletons, stubs, and the component home. This approach differs in two important aspects from the previous ones based on DotQoS, COM+, or ASP.NET/IIS. First, the PSM is not based on the UML. Second, the developers do not necessarily touch the PSM. The PSM is hidden inside the tool chain. From the viewpoint of a developer the PIM is directly mapped to source code.

The mapping to the IDL/CIDL meta model has been presented here to illustrate that a PSM is not necessarily restricted to the UML or a derivate thereof. Furthermore, the IDL/CIDL meta model is special since it has no standard graphical notation. That means there is no standard way of depicting a model based on the IDL/CIDL meta model in a modeling tool. The standard notation is a textual one as shown in subsection 10.3.5. Kafka's object diagrams have been developed for such scenarios. The object diagram shows instances of meta classes as objects and instances of meta associations as links between the objects.

Figure 12.15 provides an example. The Kafka rules transform PIM components, ports, contract types and QoS parameters into the corresponding Qedo and CORBA CCM constructs. The rules `Interface` and `Component` map PIM components and interfaces to the IDL constructs `InterfaceDef` and `ComponentDef`. The `Port` rule maps PIM ports to CORBA CCM facets. In the CORBA CCM meta model a component and its provided interfaces are connected via an instance of the `ProvidesDef` meta class. The `Contract` rule maps PIM contract types to Qedo's `ContractType` meta class. QoS parameters are translated by the `Parameter` rule. The rule searches the `ContractType` that has been created for the PIM contract by a previous rule. Then it iterates over all PIM QoS parameters. For each PIM QoS parameter it determines its storage type and searches the corresponding `IDLType`. The matching between PIM storage types and the corresponding `IDLType` uses the matching graph. The rules that build this part of the matching graph are not shown in the diagram. For each match found the `Parameter` rule adds a new `Dimension` to the `ContractType`. The rule as depicted in Figure 12.15 matches only parameters with direction increasing and milliseconds as units. Using some Python expressions the rule can be generalized. For the sake of clarity, the simplified rule without Python expressions is shown in the diagram. Finally, the `Bind` rule ties together components, interfaces, and contract types. The `Binding` is owned by the `ContractType`. The `InterfaceDef` has the role of the interaction element (see subsection 10.3.5). The `ComponentDef` acts as the context for the binding.

The example demonstrates that one Kafka rule can combine object diagrams and normal notation. The PIM search pattern uses the standard PIM notation. The PSM search and replace pattern are depicted as object diagrams.

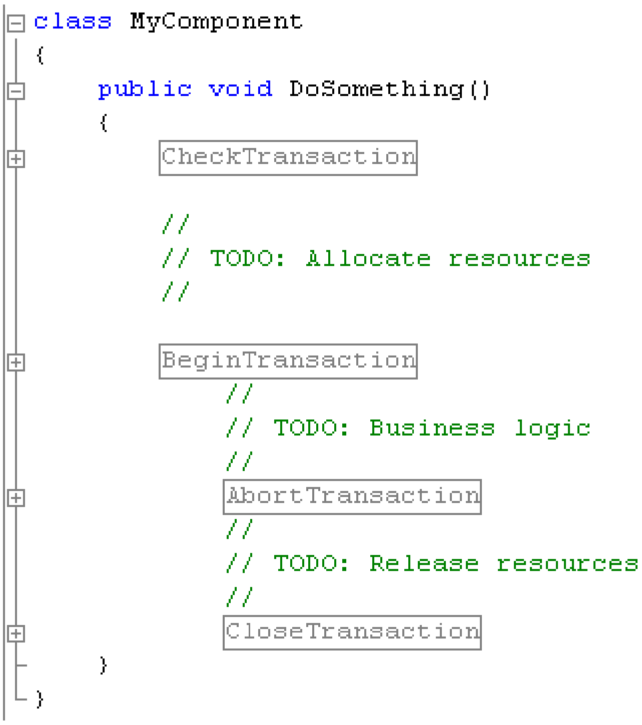


```
#region CheckTransaction
if ( !ContextUtil.IsInTransaction() )
        throw new Exception(``No Transaction Context");
#endregion

//
// TODO: Allocate resources
//

#region BeginTransaction
try
{
#endregion
        //
        // TODO: Business logic
        //
#region AbortTransaction
}
catch(Exception e)
{
        ContextUtil.setAbort();
        return;
}
finally
{
#endregion
        //
        // TODO: Release resources
        //
#region CloseTransaction
}
ContextUtil.setComplete();
#endregion
```

Figure 12.6: Implementation of `DoSomething()`

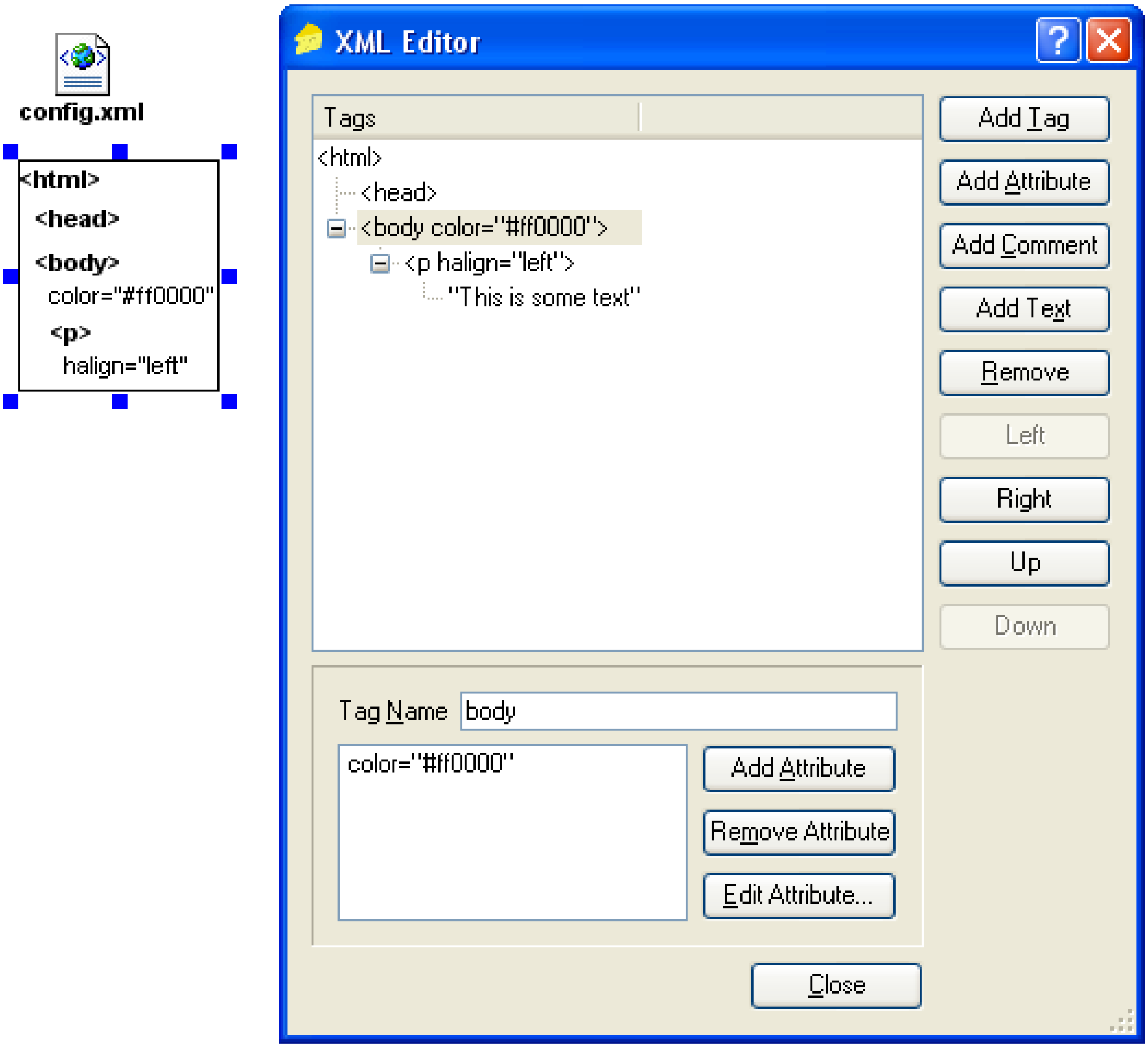


```
class MyComponent
{
    public void DoSomething()
    {
        CheckTransaction

        //
        // TODO: Allocate resources
        //

        BeginTransaction
            //
            // TODO: Business logic
            //
            AbortTransaction
            //
            // TODO: Release resources
            //
            CloseTransaction
    }
}
```

Figure 12.7: `DoSomething()` in the VS.NET editor



```
<html>
  <head>
  <body color="#ff0000">
    <p halign="left">
      "This is some text"
```

Tag Name body

color="#ff0000"

Figure 12.8: XML editing in Kase

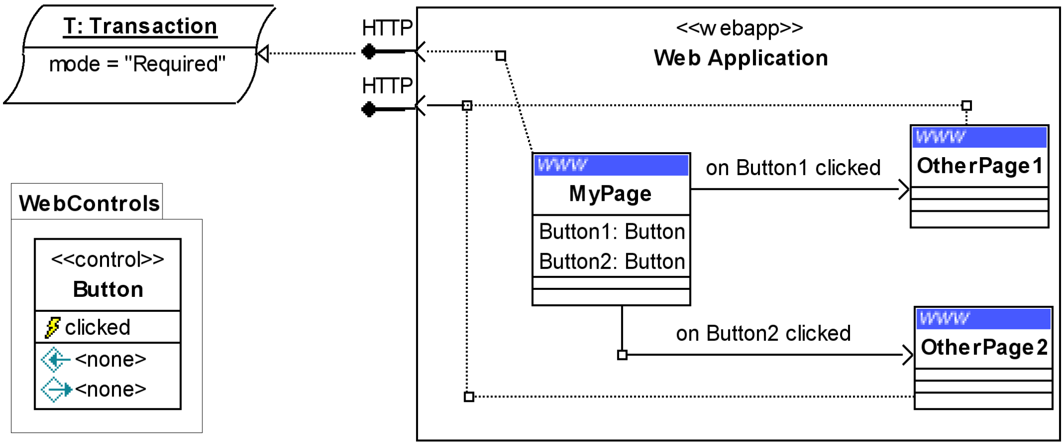


Figure 12.9: PIM of a simple web application

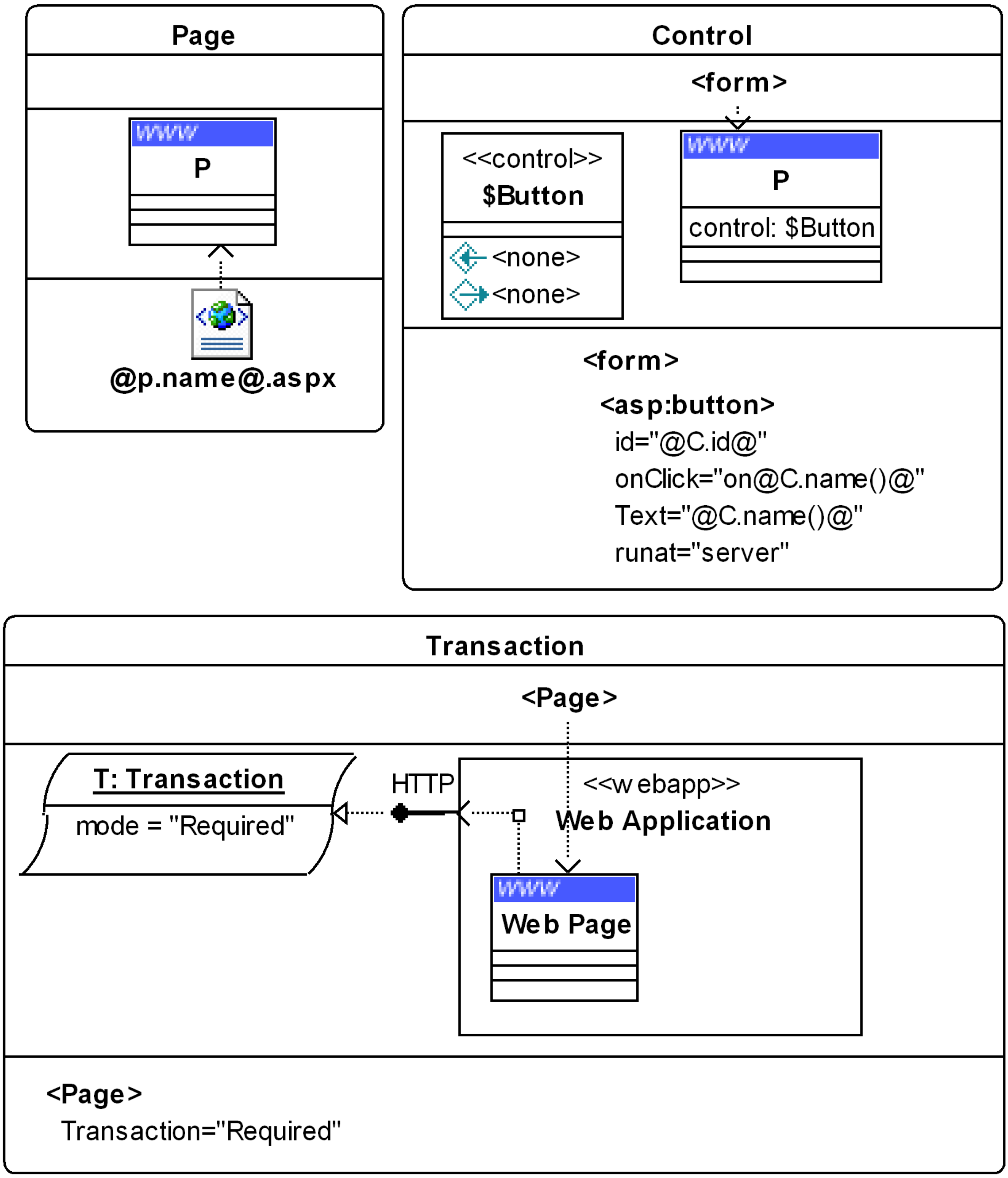


Figure 12.10: Transformation rules for web applications

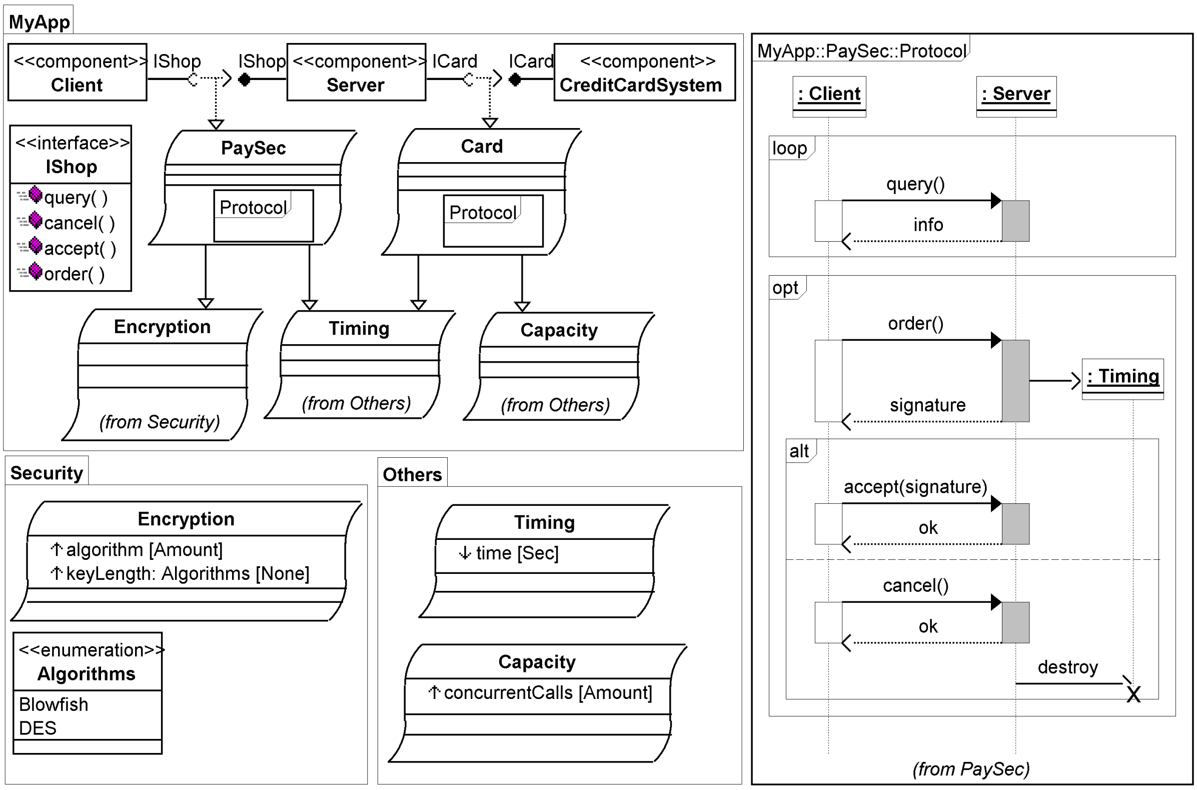


Figure 12.11: PIM for DotQoS application

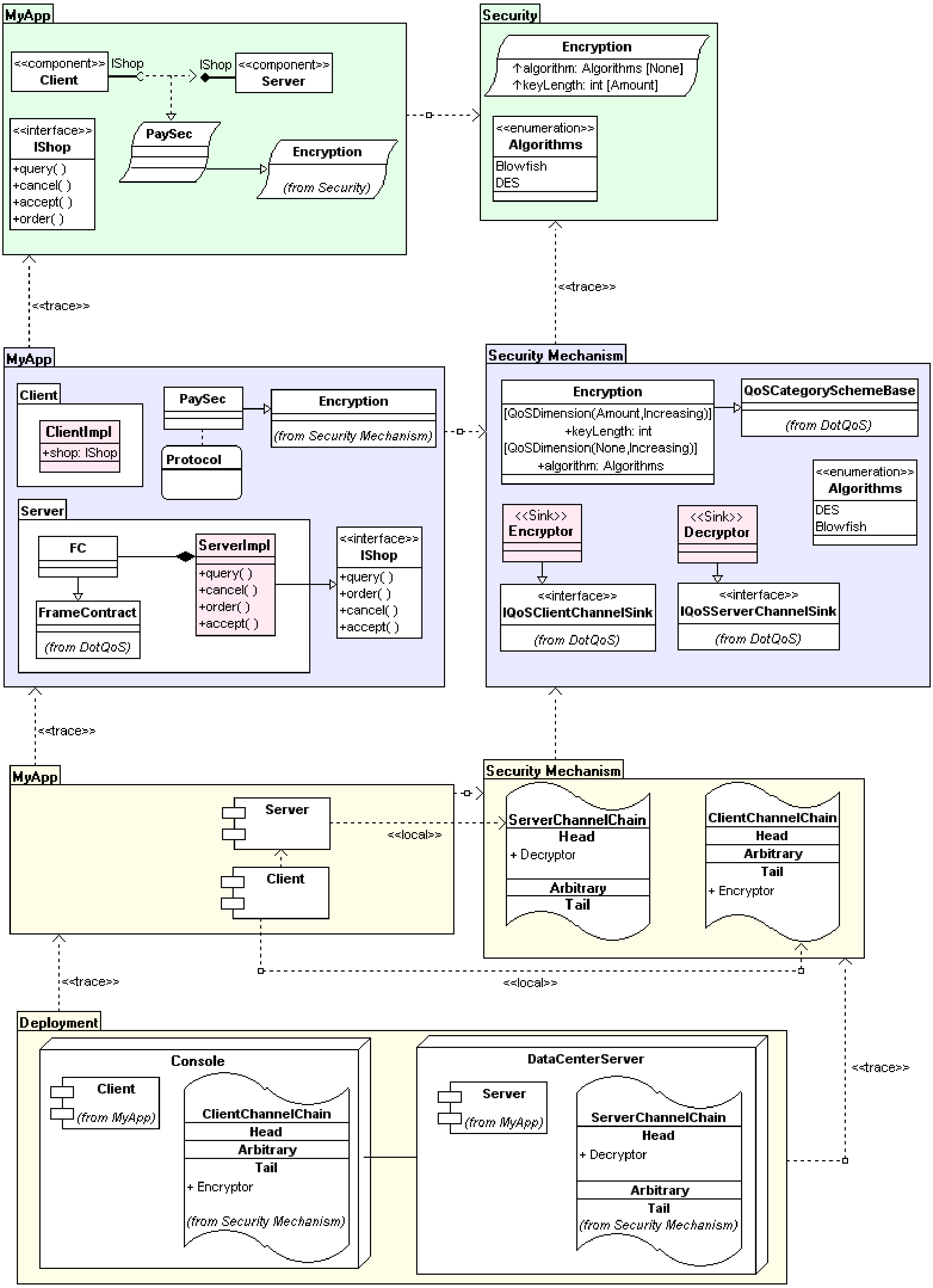


Figure 12.12: Development life-cycle of a DotQoS application

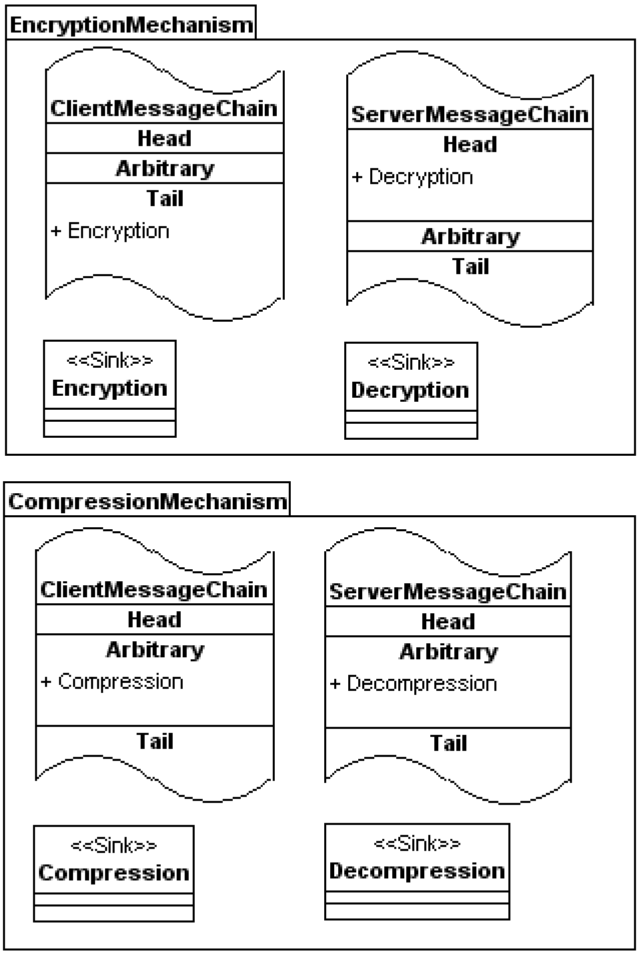


Figure 12.13: Diagram of two mechanisms

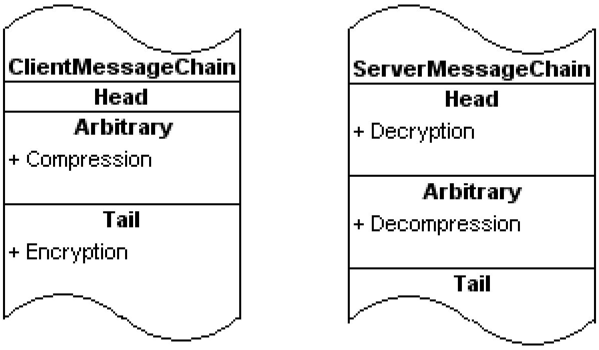


Figure 12.14: Resulting sink chain

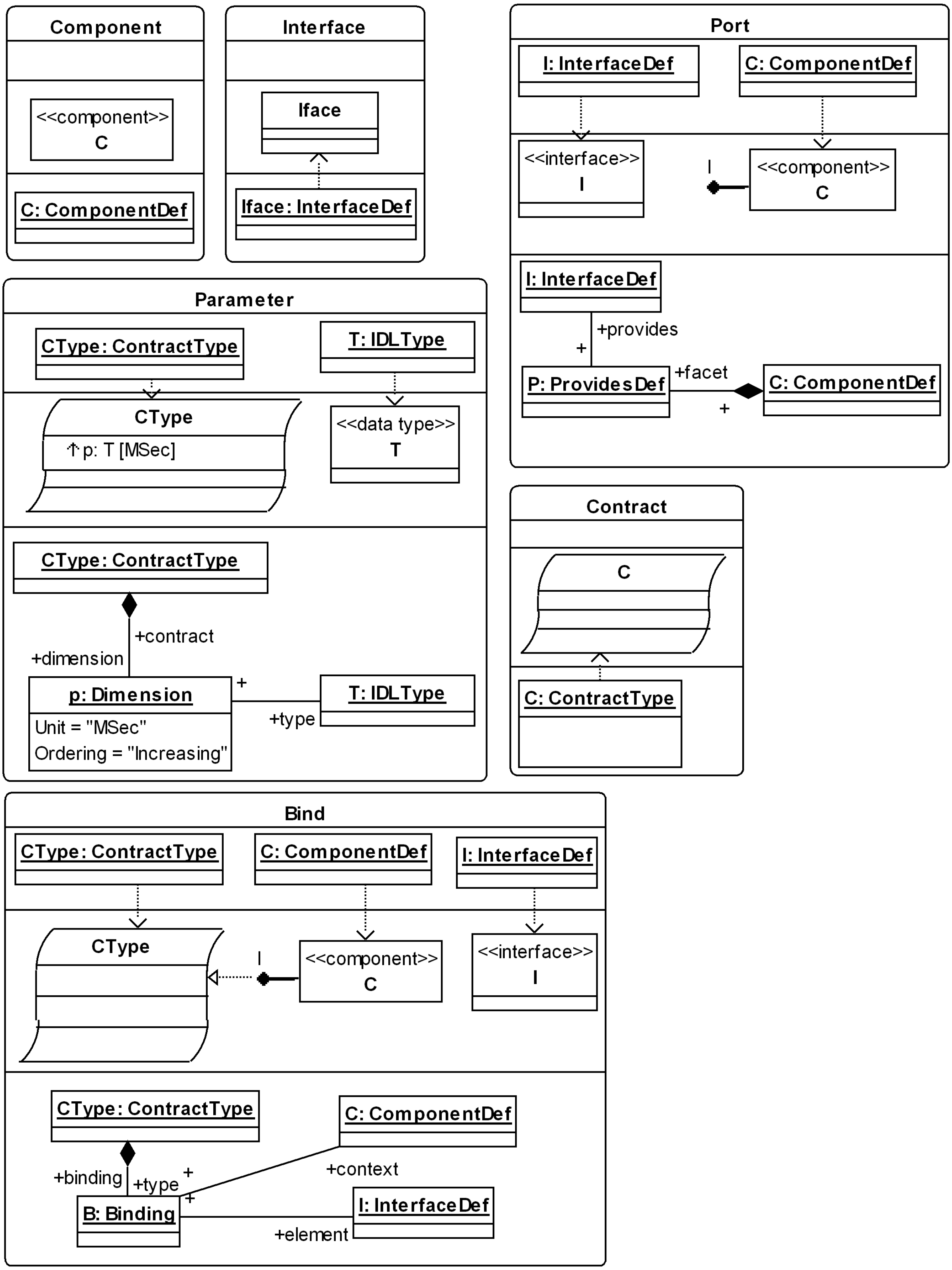


Figure 12.15: Transformation rules for Qedo

Chapter 13

Transforming QoS Contracts

This chapter discusses how QoS contracts are treated by the tool chain. The three steps required for the transformation of a PIM and its QoS contracts into a PSM are depicted in Figure 13.1.

- The first step is aware of the fact that QoS exists. However, it does not know anything about the concrete semantics of the QoS categories used in the PIM. The result of step one is a QoS-agnostic PSM.
- The second step selects the QoS aspects that have to be woven into the QoS-agnostic PSM. This requires traversing the PIM and searching for ports bound to QoS contracts.
- The third step weaves the QoS aspect with the QoS-agnostic PSM yielding the final outcome of the transformation process.

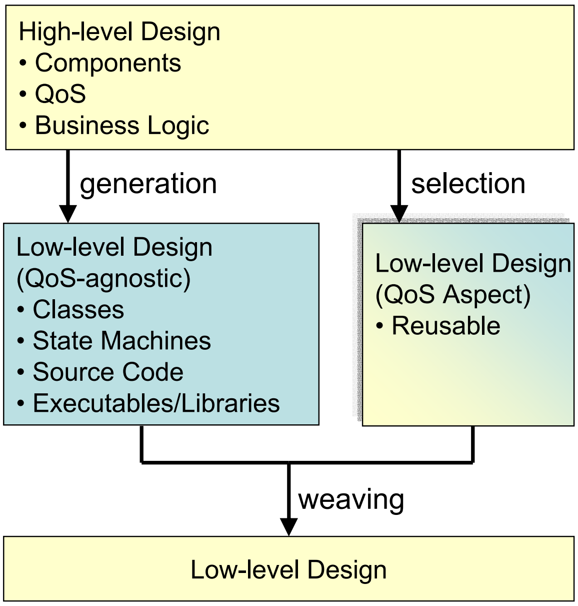


Figure 13.1: Overview of the model transformation

All three steps can be implemented via Kafka rules. This has the advantage that the set of technologies required to implement model transformation is kept down to an absolute minimum. The first of the three steps can be implemented with Kafka because Kafka has been designed primarily for this purpose: transforming a PIM into a PSM. The second step can be implemented by a Kafka rule that searches component ports that are connected with QoS contracts. Figure 13.2 shows such a rule. It searches for all component ports that realize a QoS contract type named `Authentication`. Furthermore, the rule determines the interface belonging to the port of the component. Finally, the rule adds a *transformation tag* to the component. Following rules can use this tag to determine which QoS mechanism this component will use in the PSM. This tag is especially important in the case of multi category QoS which is discussed in section 13.4. The third step – weaving the QoS aspects into the PSM – is the most complicated one. To understand its realization with Kafka, it is important to study the QoS aspects in more detail.

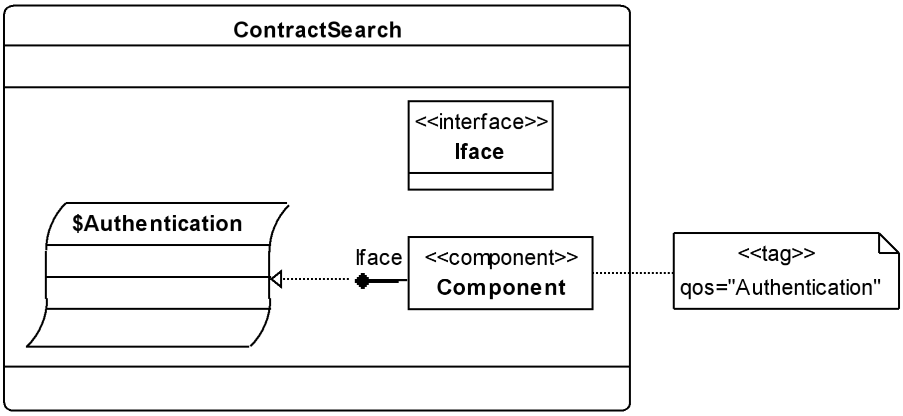


Figure 13.2: A Kafka rule for the selection of an QoS aspect

13.1 Anatomy of QoS Aspects

The box titled *QoS aspect* [WPA+03, JPWG02, SAD+02] in Figure 13.1 contains different modeling elements. Most notably the QoS aspect contains some sort of QoS mechanism or source code that activates a built-in QoS mechanism of the platform. Examples for such QoS mechanisms are message compressors, transport channels with bandwidth reservation, a load balancer etc. The middleware has to know for which component and port a certain QoS mechanism has to be used. Hence, the QoS aspect must manifest this information somehow in the final PSM. This can either happen by adding special constructs to interfaces [BG99, Rit03, UWGB03] or via deployment descriptors as in Enterprise Java Beans (EJB). The concrete solution depends entirely on the middleware.

Many QoS categories affect the application design. For example, replication or load balancing require the cooperation of the application. The application must know how to replicate its current state and how to synchronize with multiple worker processes. The QoS mechanism can just manage the worker processes. Hence, the QoS aspect must contain some kind of design pattern to add the QoS-specific design to the PSM.

Furthermore, a QoS aspect can contain a set of software artifacts (deployment descriptors, DLLs, executables, methods, etc.). For example, a QoS mechanism available as a separate library (as in DotQoS) is such a software artifact. These artifacts can show up in an implementation diagram. The implementation diagram describes which DLLs, source code files, text files, etc. are required to build (i.e. compile, link, and deploy) the application. Furthermore, the artifacts may show up in deployment diagrams (see section 12.1) since the artifacts must be installed on real machines in order of deploying the application. Thus, sophisticated QoS aspects can contain parts of an implementation diagram and they can contain constraints of the deployment as discussed in section 12.1.

Each QoS aspect is accompanied by a PIM QoS contract type. This type has an evaluation function f, parameters p, and measurands m_i. If a PIM designer uses this QoS contract type, the corresponding QoS aspect will be woven with the PSM. The mapping between QoS contract type and QoS aspect happens by name. The QoS contract type is, for example, called `Authentication`. Then the QoS aspect searches for all ports that are connected to a QoS contract type named `Authentication`.

Searching by name only seems very restrictive. However, it is in most cases sufficient, since the PIM designers are not expected to modify the QoS contract type. In the general case, it is hard to impossible to implement a QoS aspect that is independent of f, and its measurands m_i. Hence, the contract type is assumed to remain unchanged. Only the values of the parameters p are specified by PIM designers. In some cases it might be necessary to search for contracts which have special values for some parameter p_i. For example, one QoS aspect can realize availability of up to 99%, while a second QoS aspect is dedicated to systems with an availability greater than 99%. In this example, the two QoS aspects search for all components offering an availability contract with less than (respectively more than) 99%.

13.2 Aspect Diagrams

In the tool chain a QoS aspect is a set of Kafka transformation rules. The rules search for components with the appropriate contracts and generate the PSM realization of this contract. While this is satisfying from a tool chain point of view, it is not sufficient from a methodology point of view. Only very experienced developers will be able to directly formulate the appropriate Kafka rules for complex QoS categories. The solution is to introduce an intermediate step. In this intermediate step a QoS aspect developer models the structure and the behavior of the QoS aspect in a special aspect diagram. This model is not important for the tool chain. Its sole purpose is to support the developer in his creative process.

To develop any kind of generator, it is good practice to start with an example that shows the input and the output of the intended generator. Furthermore, the example should illustrate how the input influenced the output. With such an example (or a set of examples) as a starting point, it is much easier to derive a generator that covers all possible cases. A weaver is just a special kind of a generator. Hence, this principle can be applied here, too. An aspect diagram shows the structural features that a QoS aspect adds to the QoS-agnostic PSM. Furthermore, it describes how the behavior of the aspect is woven with the behavior of the QoS-agnostic PSM.

Figure 13.3 provides an example. The right half of the figure contains a PIM and a PSM model. The PSM shows (partially) the QoS-agnostic PSM that would be generated from the

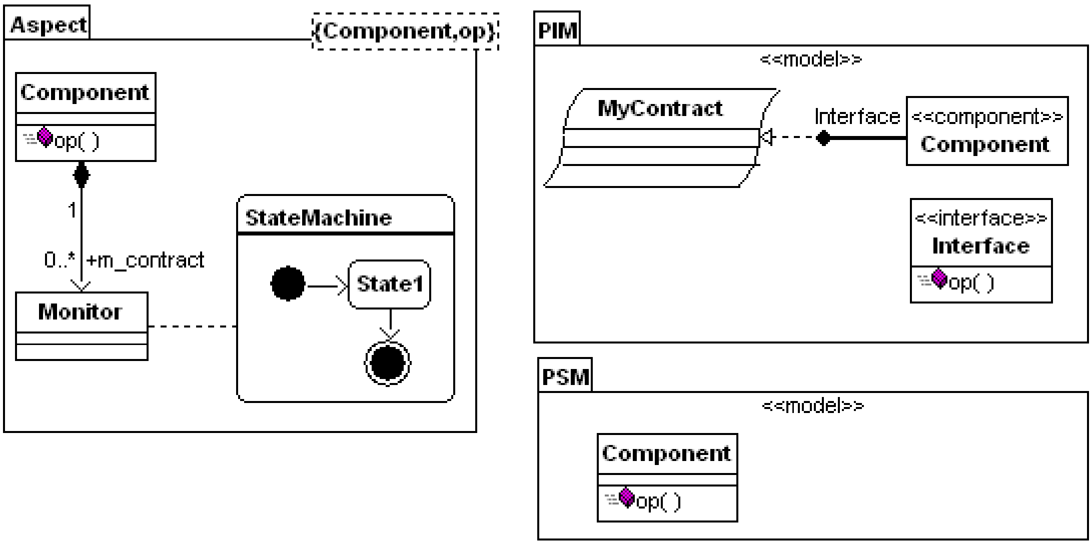


Figure 13.3: An aspect diagram

PIM. Both models are in no way complete. They just contain the modeling elements that are important for the QoS aspect. The left half of Figure 13.3 shows the aspect diagram. From a UML point of view, an aspect diagram is a package template. The diagram shows in how far the QoS-agnostic PSM is extended and modified. In this example, a monitor class, a composition, and a state machine is added by the QoS aspect to the QoS-agnostic PSM. The dashed box in the upper right corner of the aspect diagram shows the join points, or template parameters in UML parlance. The example has two of them called `Component` and `op`. Modeling elements of the same name appear in the aspect diagram. These modeling elements stem from the QoS-agnostic PSM. Hence, the template parameters are useful to define how the model elements of the aspect are glued with the model elements of the QoS-agnostic PSM.

13.2.1 Aspect Behavior

An aspect diagram can model how the behavior of the aspect is woven with the behavior of the QoS-agnostic PSM. An interaction diagram is used for this purpose. This diagram shows the interaction between QoS-agnostic objects and QoS-aware objects. The purpose of this diagram is two fold. First, it shows the QoS-agnostic behavior that the aspect expects. Second, it shows in which situations the QoS-agnostic behavior is changed by the aspect.

Figure 13.4 illustrates this concept. The aspect models how a timeout is woven into a component that acts as client to some kind of FTP-like service. The aspects expects the class `Component` and the interface `FTP-Service` to exist in the QoS-agnostic PSM. Furthermore, it expects that a set of methods `connect, abort, login, pwd,` and `list` exist. The aspect adds the `Timer` class. The interaction diagram embedded in the aspect diagram shows how the behavior of the QoS-agnostic PSM is altered. If some caller invokes the `connect` method, the aspect enforces the creation of a `Timer` object. In the following, two threads execute in parallel. In the first thread, the component follows its QoS-agnostic behavior. It calls several

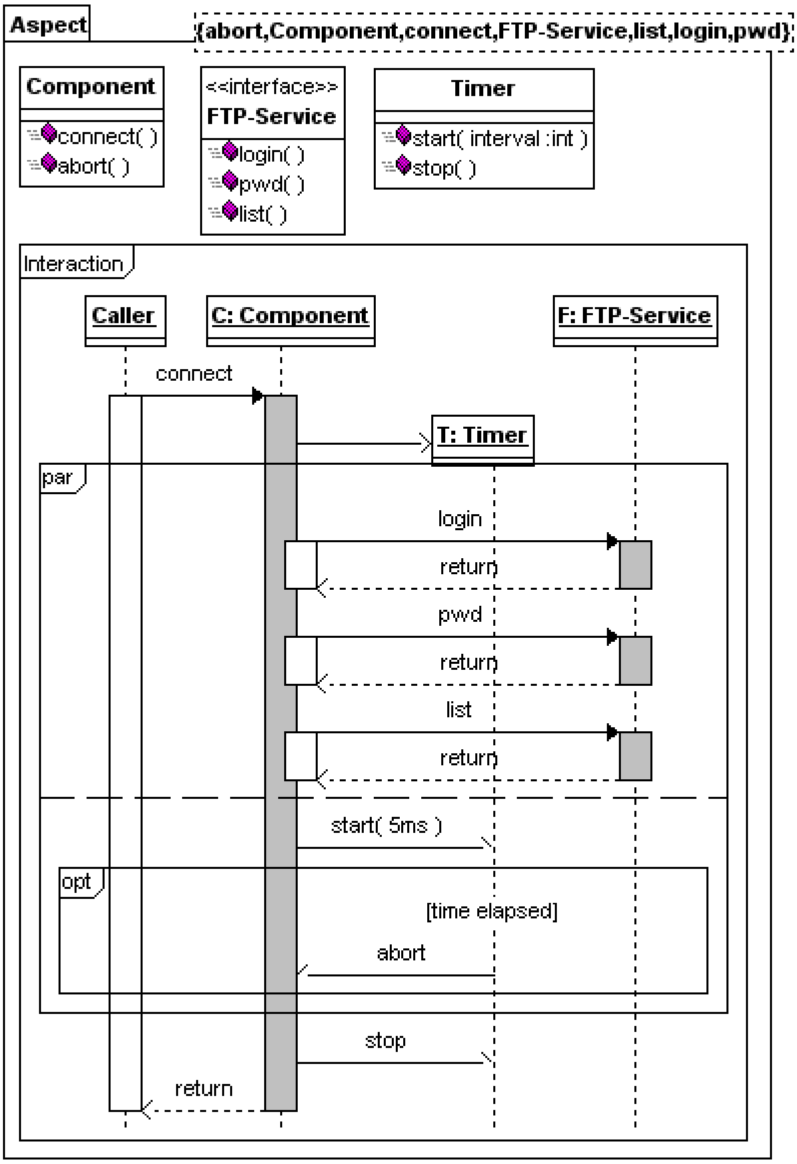


Figure 13.4: Aspect with interaction diagram

methods of the `FTP-Service`. In the second thread, the `start` method of the timer is called. If a timeout occurs, the timer invokes the `abort` method of the component. If the communication with the FTP-Service finishes in time, the two threads join and the `stop` method of the timer is called.

In some cases it is not obvious, which method calls are expected from the QoS-agnostic PSM and which are introduced by the aspect. This can be solved in two ways. First, another interaction diagram models the behavior of the QoS-agnostic PSM. In this case, the aspect behavior is the difference between the two interaction diagrams. The other option is to mark the aspect-specific method calls either with a special color or an asterix in front of the method name, i.e `*stop` instead of `stop`.

13.2.2 Transition to Kafka

The transition from an aspect diagram to a set of Kafka rules can be quite easy. Figure 13.3 shows that the combination of PIM model, QoS-agnostic PSM and QoS-aspect correspond closely with the three compartments of a Kafka rule. The PIM model will end up in the PIM search compartment of Kafka rules. The QoS-agnostic PSM will find its way in the PSM search compartment of Kafka rules. Finally, the model elements of the QoS-aspect can be moved into the replace compartments. If the QoS aspect contains an interaction diagram as shown in Figure 13.4, a bit more cleverness is required to alter the behavior of the woven PSM via Kafka rules. In this case the rules must either inject source code in the implementation body of methods, or behavioral diagrams of the QoS-agnostic PSM must be modified by Kafka rules.

Usually, an aspect diagram will result in a set of Kafka rules. This is due to multiplicity relationships. For example, a QoS-agnostic PSM contains a set of classes and each class a set of methods. The aspect wants to add a new attribute to every class and some source code lines to every method. This example requires two Kafka rules. One rule is applied to every class. The second rule is applied to every method.

13.3 Weaving QoS Aspects

The first challenge for aspect weaving (in the sense of AOP) is to determine *join points*. Join points define the possible hooks in a system where an aspect could be hooked in. For example, an aspect may intend to do something whenever a new instance of a class is created. Hence, the aspect would hook itself into the class' constructor.

In Kafka each model element of the PSM and the PIM can be treated as a join point. A QoS aspect can hook itself up to any model element in order to change or even remove it. In AOP languages – such as AspectJ [Asp03] – aspects define so called *point cuts*. A point cut is a pattern that is matched against all possible join points. If a point cut matches a join point (or a set of join points), the aspect is hooked up to this join point.

In Kafka the PIM search pattern and the PSM search pattern can be treated as point cuts. The search patterns are matched against all PIM and PSM model elements (i.e. the join points). If a set of model elements matches the pattern, the QoS aspect is injected in the design.

The example in Figure 13.5 illustrates this idea. The rule searches for a PIM component and its realization in the PSM. The tag on the right-hand side of the rule is used to limit the matched components to those that realize the authentication contract. This complements the tag found in Figure 13.2. If a match is found, the method `DoSomething()` is injected into the design.

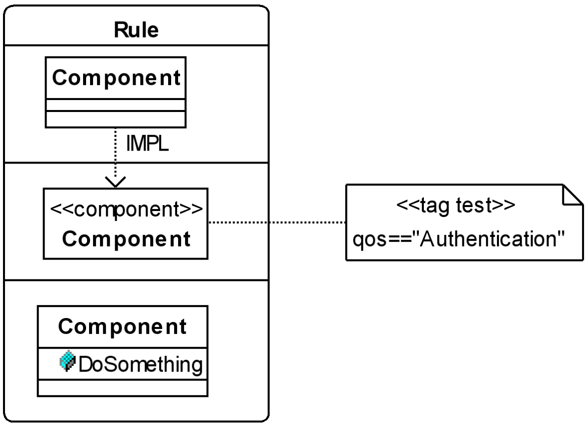


Figure 13.5: A Kafka rule that injects a new method

PSMs – especially when they are based on UML – can contain source code. For example, the body of a method – i.e. source code – can be embedded in the model. Furthermore, state machines contain actions which can have associated source code, too. Weaving QoS aspects can include the modification of such source code. Typical tasks are to prepend some lines of source code or to insert some code in front of a `return` statement. In Kafka the modification of any kind of text, including source code, or numbers, timeouts for example, are performed via embedded Python statements.

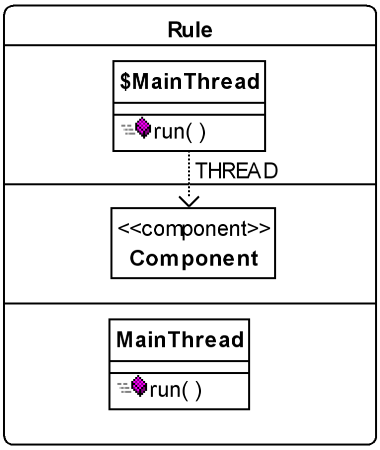
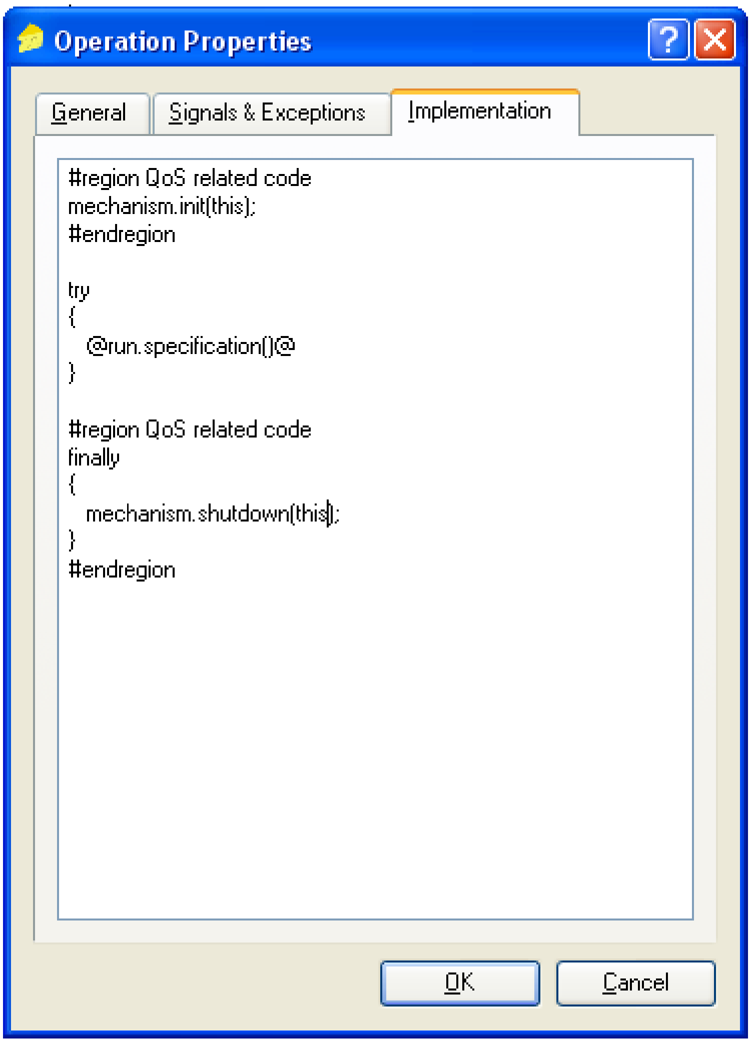


Figure 13.6: A Kafka rule that modifies the implementation of the `run()` method

For example, the transformation rule in Figure 13.6 injects source code in the `run()` method of a class named `MainThread`. Figure 13.6 looks similar to Figure 13.5. The interesting part is the body of `run()` method in the rule's replace pattern. This body is depicted in Figure 13.7. The

Python statement `@run.specification()@` is replaced by the previous body of `run()`. The rule prepends some lines of source code to the implementation of `run()` and prepends some other lines.



```
#region QoS related code
mechanism.init(this);
#endregion

try
{
  @run.specification()@
}

#region QoS related code
finally
{
  mechanism.shutdown(this);
}
#endregion
```

Figure 13.7: The method body of the `run()` method in the rule's replace pattern

Another way of weaving new behavior into a class is to modify its behavioral diagrams, for example a state machine. The two rules in Figure 13.8 add a new QoS-related state `QoSBroke` to a state machine. Whenever a QoS contract is violated, the state `QoSBroke` is entered. The only transition away from `QoSBroke` leads to the state machine's final state. Rule `R1` adds `QoSBroke` and connects it with the final state. Rule `R2` searches all states of the state machine and connects them with `QoSBroke`. The precondition is used to avoid a transition that starts and ends in the `QoSBroke` state. The prefix `R1::` is used to refer to elements that have been created or matched by rule `R1`.

It would not be possible to merge both rules into one. `R1` is executed once for every state machine. `R2` is executed once for every state in each state machine matched by `R1`.

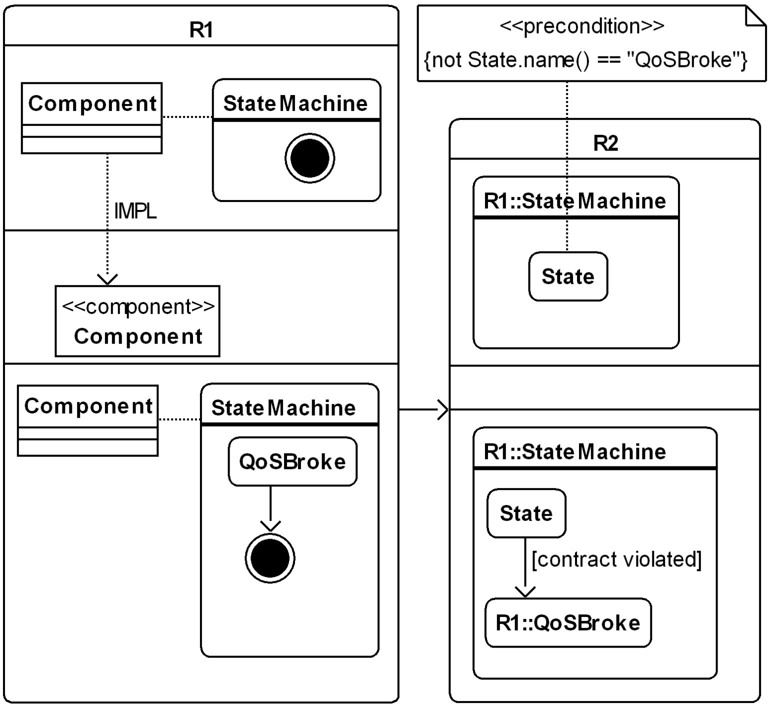


Figure 13.8: Modification of a state machine

AspectJ supports regular expressions in point cuts. Thus, it is possible to catch the invocation of every method that has a name starting with `set`. The same solution can be used in Kafka rules. Names of model elements in a rule's search patterns are treated as regular expressions. The PSM search pattern in Figure 13.9 searches all setter methods of a class using the regular expression `set[A-Z].*` [Fri02].

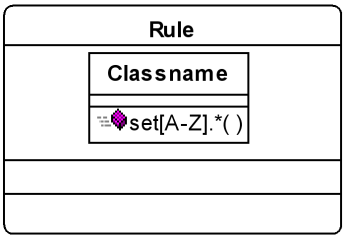


Figure 13.9: A rule searching for setter methods

The weaving of QoS aspects and the weaving of AspectJ-style aspects can be compared to a certain degree only. AspectJ developers do not have to care about the possible ugliness of the woven source code. It is directly passed on to the compiler. Developers will only face the woven source code when they watch the application via a debugger.

Weaving QoS aspects into the PSM has the intention of injecting design aspects into the PSM. The woven PSM will be visible to developers. They must understand and work with the woven design. Therefore, weaving QoS aspects targets the design phase. Weaving aspects in the sense of AspectJ targets the runtime behavior. Any obfuscation of the design is acceptable as long as the expected runtime behavior is guaranteed.

Thus, weaving QoS aspects and AspectJ aim at different problems. They have in common the problem of cross cutting. AspectJ-style aspects cannot be isolated in the source code using Java language constructs. QoS aspects cannot be isolated in a UML model using UML constructs. Both approaches have some similarities in the way they define aspects, join points and point cuts. However, the weaving of QoS aspects as presented in this thesis is not a direct extension of AspectJ to the design phase. Therefore, Kafka should not be treated as an AOP language, although it can be utilized to overcome the problem of cross cutting in certain settings.

13.4 Multi-Category QoS

A problem may arise if one component port features several QoS categories. In this case multiple design patterns may be applied to the PSM representation of the port. It could happen that the first design pattern modifies the PSM in such a way that the second one cannot be applied any more.

There are two ways of solving the problem. In some cases it is just a question of the ordering of transformations. The outcome of the transformations

$$\begin{gathered} G \Rightarrow_a G_1 \Rightarrow_b H_1 \\ G \Rightarrow_b G_2 \Rightarrow_a H_2 \end{gathered}$$

may differ, hence H_1 and H_2 are not necessarily equal. Ordering can be achieved via the orchestration of Kafka rules.

If ordering cannot solve the problem then a special set of transformation rules is needed to handle this special combination of QoS categories. This sort of problem is inherent to all generic multi-category QoS approaches because some QoS categories are not completely orthogonal to each other. A related problem appears in the DotQoS PSM when it comes to the ordering of message sinks (see subsection 12.4.1). The solution works along the same lines.

If multi-category QoS requires special mechanisms to deal with a special combination of QoS categories, Kafka rules can be utilized to detect components that require special QoS mechanisms. Figure 13.10 shows a possible solution. The first rule `R1` searches for components that realize the QoS contract `Type1`. Rule `R2` does the same for `Type2`. Finally, rule `R3` searches for multi-category QoS components. The solution depends on the tags. In case of multi-category QoS the `mechanism` tag's value will be `1&2`. Subsequent rules search for the `mechanism` tag and its value and apply the indicated QoS mechanism.

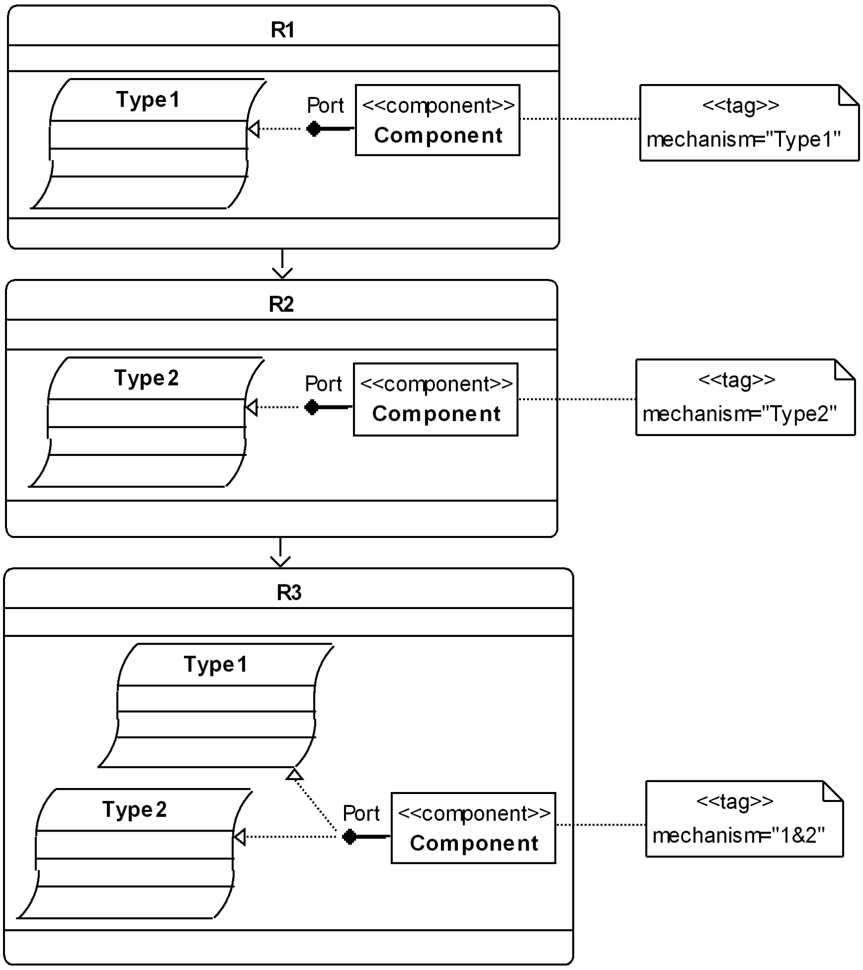


Figure 13.10: Collision detection for multi-category QoS

Part IV

Outlook & Conclusions

Chapter 14

Conclusions

This chapter is split in two sections. The first section reviews the findings of this thesis and the goals achieved. The second section discusses the pros and cons of the chosen approach.

14.1 Review

This thesis has shown the importance of models for software development, especially in the context of distributed and QoS-enabled applications. Models allow developers to work on a higher level of abstraction. Hence, developers can concentrate on their application, the business logic and the QoS requirements. Their attention is not distracted with implementation-specific decisions and optimizations. Model-driven development can at the same time provide a very convenient way of programming and still produce optimized code. Without model-driven development, these two goals are often mutual exclusive. For example, if a set of classes in an object-oriented language provide a convenient level of abstraction for implementation, it is quite likely the classes waste time and memory space to provide this abstraction. By introducing two models, one being platform (and implementation) independent and one being platform specific, it is possible to have a convenient level of abstraction at the PIM and an optimized realization in the PSM.

During the work on this thesis it turned out that tooling is an essential problem of model-driven development. Developers have to use a couple of new tools – especially modeling, editing, and model transformation tools – in addition to their traditional tool chain, i.e. source code editor, compiler, debugger. The advantages of model-driven development can vanish, if developers spend too much time with badly designed tools. Therefore, the usability of the model editing tool and an intuitive modeling language are important. Model transformation has proven to be very hard to address. Model transformation approaches based solely on imperative or functional languages complicate the implementation of transformers too much. Developers who are not real experts in meta models cannot quickly implement even simple transformers this way. Hence, a new and more user-friendly approach has been developed throughout this thesis. In summary, the following three building blocks supporting model-driven development have been presented in the previous chapters.

PIQML is a high-level platform independent modeling language (PIQML) based on UML. It supports component-based modeling of distributed applications. Components communicate via ports and ports may be subject to QoS contracts. Furthermore, PIQML supports the task concept, i.e. instead of attaching QoS contracts to entire ports, they can be attached to a special interaction patterns, i.e. tasks.

Kase is an easy to use model editor for PIMs, PSMs, and Kafka diagrams. In contrast to other modeling tools, Kase does not have a fixed meta model and notation. Both can be extended via plugins. This way it is not necessary to develop a new modeling tool for new versions of PIQML or the UML. Kase has a built-in Python interpreter for inspection and transformation of models. This tight integration improves model transformation is a way that is difficult to achieve with more loosely coupled XMI-based solutions.

Kafka is a visual rule-based model transformation language. The advantage of Kafka is that it builds on a fairly simple concept: search and replace. The notation used for the search and replace patterns builds on the notation used for the PIMs and PSMs. If a developer can read PIMs and PSMs, Kafka rules are easy to read and construct, too. Transformation rules in Kafka are in most cases easier to construct than their Python counterparts. In cases where an imperative language is better suited, Python expressions can be embedded in Kafka rules. Hence, Kafka is symbiosis of rule-based and imperative languages.

This thesis has shown that model-driven development can improve the creation of complex QoS-enabled distributed applications in terms of quality and development time. The following list highlights the improvements resulting from the model-driven approach.

Make design decisions explicit: The PIM shows information that is otherwise only implicitly contained in the implementation of an application. Most notably, the QoS contracts and tasks of components are explicitly modeled in the PIM. In the PSM and especially in the source code, tasks are mapped to state machines and QoS contracts are mapped to code utilizing QoS mechanisms of the target platform. Hence, it is very hard to extract the information presented in the PIM from PSM and source code. Furthermore, model transformation can derive an implementation (or at least a skeleton) from the PIM. This way the PIM is not just extra work required for documentation. It becomes an active asset of the development process.

Abstraction from middleware specific details: The PIM is by definition platform independent. Therefore, it allows abstracting from details of the target platform. Designers can concentrate on the core of their problem: the application. Concentrating on the application and ignoring platform details helps preventing “premature optimization”which “is the root of all evil”[Knu92]. Furthermore, the decision for a certain target platform can be deferred. Development teams can choose the appropriate target platform *after* they finished the first iteration of the PIM.

Tasks help to improve QoS contracts: For enterprise applications the execution time of a certain method is not as important as for embedded real-time systems. Hence, providing QoS contracts for single method invocations is not an ideal solution for enterprise applications. Attaching QoS contracts to tasks is more promising. This thesis has proven that the worst case runtime of tasks can be better approximated if the QoS contract is attached to the task not not to single methods. Therefore, the introduction of tasks is not only a syntactic improvement. Using tasks, QoS contracts about timeliness can be closer to the real worst case runtime required for the complete task.

Automation of recurrent work : Model transformation automates boring and recurrent – i.e. error prone – work. The model transformer can, for example, insert code pieces or special states in the implementation of many methods or state machines. Doing this manually requires time and has the inherent danger that a developer forgets to insert the code or design artifact in all relevant places. Furthermore, the model transformer can automatically update these code pieces and design artifacts when the PIM changes. That means, the relevant information is changed in a single place and the code pieces and design artifacts scattered across the PSM and source code are automatically updated. Hence, a model transformer does not only save work, it can even help to keep consistency in the PSM and source code.

Encapsulate expert knowledge: The knowledge about how to map a QoS contract onto a target platform is encapsulated in QoS aspects. Hence, expert knowledge can be separated from business logic and packaged in a tool. Less skilled developers can simply use an existing contract type in their PIM. The model transformer automatically weaves the corresponding QoS aspect with the QoS-agnostic PSM. Hence, the less skilled developers do not need to understand in depth how QoS is realized on the target platform in order of implementing a QoS-enabled application.

Reuse: In a pure source code oriented development it would not be possible to reuse QoS-related code and design. On the implementation level QoS is known to be cross cutting [BG98, HBG$^+$99, PLS$^+$00a]. The QoS-related code cannot be encapsulated in a class or a behavioral UML diagram. However, it is very difficult to reuse something that cannot even be encapsulated. The case is different for the approach presented in this thesis. QoS aspects allow separating QoS-related design and code pieces. Over time companies can build a library of contract types and associated QoS aspects. Thus, knowledge that has been gained in previous projects can be directly applied to new projects.

Support for deployment: Model transformers can create constraints on the deployment. These constraints are evaluated by the modeling tool when developers create the deployment diagrams. If a constraint is violated, the modeling tool prompts the user. This is very important since the realization of a QoS contract is not always possible without dedicated hardware support. For example, the best load balancing does not help if the network connection between the machines is too weak or if the servers are not potent enough. Using constraints on the deployment diagrams, it is possible that the QoS contracts of the PIM influence the deployment.

14.2 Criticism

14.2.1 Overhead

Model-driven development imposes a certain overhead. New tools must be installed and mastered in every day work. This involves costs and training. For large scale projects these investments will pay off quickly, especially if transformers developed during one project can be reused in new projects. The case might be different for small scale projects. The fewer people work on a project, the worse is the overhead introduced by a process.

Implementing transformers is a time consuming tasks. Hence, it might be tempting to skip this step and hand craft the transformation. During the first iteration this is for sure the fastest solution. In subsequent iterations the transformer will pay off since it keeps consistency between PIM and PSM and gathers information in a central place. However, writing a transformer is an upfront investment. Agile development tries to minimize such upfront investments. Future will show whether model-driven development can be combined with agile development and extreme programming. [Fow01] discusses whether modeling can be combined with extreme programming at all. His conclusions are that UML can be used where it proves to be immediately useful, especially for communication between developers.

14.2.2 Notation

For many computer scientists the term modeling implies automatically a graphical notation. It would as well be possible to use a textual notation for models. The meta model (called abstract syntax tree) can theoretically be combined with any kind of notation: 2D, 3D, or textual. Consequently, the OMG published lately a standard for deriving a textual notation from meta models [OMG02c]. From a scientific point of view it is very difficult to judge which solution is the best one. A solution is good if humans can easily and quickly perceive the information depicted with a certain notation. Furthermore, an adequate tool for editing the model is required. Science did not yet come up with a model of how humans perceive information. Hence, it is difficult to proof the quality of notations in a formal way.

If the solution is a textual language, it is still hard to find a *good* textual language. Even usability tests with textual programming languages are very difficult. Anders Hejlsberg (father of Turbo Pascal and C#) states in an interview [HEV03] that it is not feasible to make usability studies for textual programming languages. Hejlsberg points out that good language design is a matter of “aesthetics”and “good taste”, but “good taste is extremely subjective and hard to define”[HEV03].

However, a definite advantage of every textual notation is that you can read the text and afterwards you can be sure that you read all information. The case is different for graphical notations. A diagram is a collage of boxes, lines, and small text fragments. A diagram cannot be read top-down. The eye follows the lines from one box to the other. In large and complex diagrams it can easily happen that even careful diagram readers miss important information. This claim can be easily proofed at the example of Hieronymus Bosch’s famous triptych “The Garden of Earthly Delight”[Bos04]. The picture has so much detail and so many actors, that it is very hard not to miss an important detail. Therefore, it might be possible that every graphical notation has a complexity threshold. In this case, every diagram exceeding the threshold would be worse than a good textual notation.

Furthermore, the concrete notation of UML has several drawbacks. PIQML being UML-based inherited these drawbacks. UML does not make use of layout. Except for message sequence diagrams designers can arrange classes, objects, actors, use cases, etc. as they desire and they can draw lines in really wired ways. The result is that designers can draw totally unintuitive diagrams just by messing up the layout. Some textual languages (for example, Python) address this problem. Layout (i.e. white spaces in the text) has a meaning. This way, Python code has always a unique layout. C/C++ is the counter example. These languages ignore layout (i.e. white spaces). Another problem of layout is that tools usually cannot render good looking diagrams. If a new model is generated (either via a model transformer or reverse engineered from source code), the user spends much time on arranging the boxes and lines on the screen. Fewer degrees of freedom in the layout could ease automatic layout and thus improve the interaction with the modeling tool. Furthermore, diagrams would instantly get a more unique look.

Chapter 15

Outlook

15.1 Extension of the Tool Chain

The presented tool chain covers the entire development cycle from PIM modeling to source code generation and deployment. However, some additional tools can be envisaged.

15.1.1 Model Checking

Experience shows that design errors are easier to resolve if they are detected early. Hence, it is desirable to perform some checking on the PIM. With respect to QoS, a model checker could search for unrealistic PIM QoS contracts. For example, a component in the PIM offers 200 concurrent client-components to perform a certain task. The model checker could compute the resulting number of expected incoming method calls per minute. If the target platform is known, the model checker could estimate whether the computed frequency of method calls can realistically be handled. Even if a realization may be possible, it may be very expensive, because an entire server farm is required to realize the amount of incoming method calls. The model checker could find such contracts and tell the designer about the costs involved.

15.1.2 Simulations

QoS-enabled distributed applications can adapt to a changing environment. Hence, a simulator could be used to see already in the PIM how QoS contracts are adapted and re-negotiated when the (simulated) environment changes. Such a simulator would require a very detailed PIM. If the PIM does not specify the behavior detailed enough then it cannot be simulated. As [TPV03] stated, simulations can only be performed when the PIM is executable.

15.1.3 Testing

Testing is a very complex and time consuming task. Integrating testing in model-driven development could significantly speed up the implementation of test cases and could help to understand the results of tests. The lines of code required for test cases can exceed the lines of code required for the implementation of the tested component. Hence, it is desirable to use the benefits of model-driven development for the generation of test case implementations. Developers could model their test cases on a high level of abstraction, e.g. the PIM. The model transformation tool could then generate the implementation or at least a skeleton implementation for each test case. This would save time and money.

Once the tests are executed, the results are gathered in a testing report. A special back-annotation tool could parse the report and back-annotate the results to the models. For example, a developer could then see in the PIM which QoS contract has been violated during the tests. This eases the interpretation of testing results.

15.1.4 Monitoring and Administration

Administrating a large scale distributed system is a difficult task. Systems composed of many components, which rely on a large set of QoS contracts, add even more complexity to administration. The administrator should be able to monitor important contracts and eventually he should be able to adapt them manually. This requires special tool support. One possible solution is to use the PIM as a basis for such a tool. The PIM shows graphically all components and their QoS contracts. The results of QoS contract monitoring could be displayed life in the PIM. Hence, the PIM is used to monitor the running application. This idea is comparable to the test results back-annotation tool. The difference is that the monitoring happens in real-time.

15.1.5 Refactoring

Model refactoring [FBB$^+$99, SPTJ01] is another application of model transformation. The difference to an MDA transformation is that the output is a variation of the input. In contrast, the output of an MDA transformation, i.e. the PSM, is created from scratch during a transformation. Kafka could be used for refactoring, too. A refactoring rule would only contain two compartments. One compartment contains a search pattern and the other one a replace pattern.

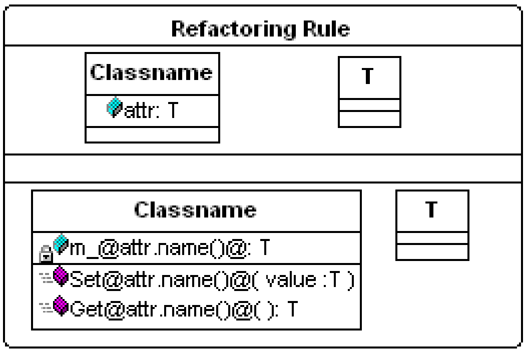


Figure 15.1: Kafka used for refactoring

Figure 15.1 illustrates this idea. The presented rule searches for public attributes of classes. If found, a getter and setter method is added, the source code for both is generated (not visible in the figure), and the attribute is renamed and its visibility is changed to private. First tests of this idea have already been successful.

15.2 Modeling Languages

In its current incarnation the platform independent modeling language PIQML allows for modeling of components, their interaction (e.g. tasks), and their QoS contracts. PIQML itself is not targeted at a special application domain, hence, PIQML components are still black boxes. Currently, the only way of modeling their internals is to recursively decompose a component in sub-components. In the context of a European IST project PIQML has been extended with concepts for the modeling of three-tier web applications. In the future, PIQML could be adapted to other application domains such as web services. Furthermore, PIQML could be extended with more behavioral concepts until it reaches a state where it can be treated as an *executable model.*

15.2.1 Adaptation Policies

Currently, PIQML allows for modeling of QoS contract types and concrete QoS contracts. Furthermore, the interdependencies of contracts can be expressed. PIQML does not allow modeling the policies that are used to update contracts. For example, if a contract is not sufficient anymore and it is renegotiated, a policy can specify whether the negotiation tries to get a higher QoS level than actually required. This policy avoids that the system is flooded with renegotiations. If the model contains such policies, negotiation algorithms could be automatically derived from the model. Adaptation policies in the PIM would replace the stereotypes ≪mandatory≫, ≪proactive≫, ≪reactive≫, ≪monitoring≫, and ≪static≫. In the current PIQML they allow for a rough categorization of the adaptation and negotiation policy.

15.2.2 New Application Domains

PIQML aims at component-based distributed enterprise applications. Currently, the principles developed in this thesis are applied to a different application domain - robot control. Toy-robots such as Lego Mindstorm [Leg03] and Fischertechnik Mobile Robots [Fis03] are ideal for teaching computer science to pupils and students. These robots have a processor, RAM, sensors, motors, and some kind of communication device (IR or serial cable). The goal of the current development is to develop a modeling language for these robots. The same modeling language is used for programming robots built on construction kits of different hardware vendors, i.e. Lego Mindstorm and Fischertechnik. Robots can be treated as real-time systems. If the control program is too slow, the robot can crash against a wall or fall down from a table. Hence, timing related QoS properties are utterly important. QoS properties are not modeled as separate contracts in this application domain. Therefore, the separation between functionality and QoS contracts is not upheld by this language. Instead, timing related concepts became an integral part of the new modeling language.

In contrast to PIQML the robot modeling language has no similarities with UML, neither with regards to the meta model nor notation-wise. Figure 15.2 shows an example. The modeled robot control program consists of three concurrent processes. Processes can communicate via events. Each process is implemented using an easy to learn imperative language and some communication primitives. A process can emit event or wait for a set of events. The modeling

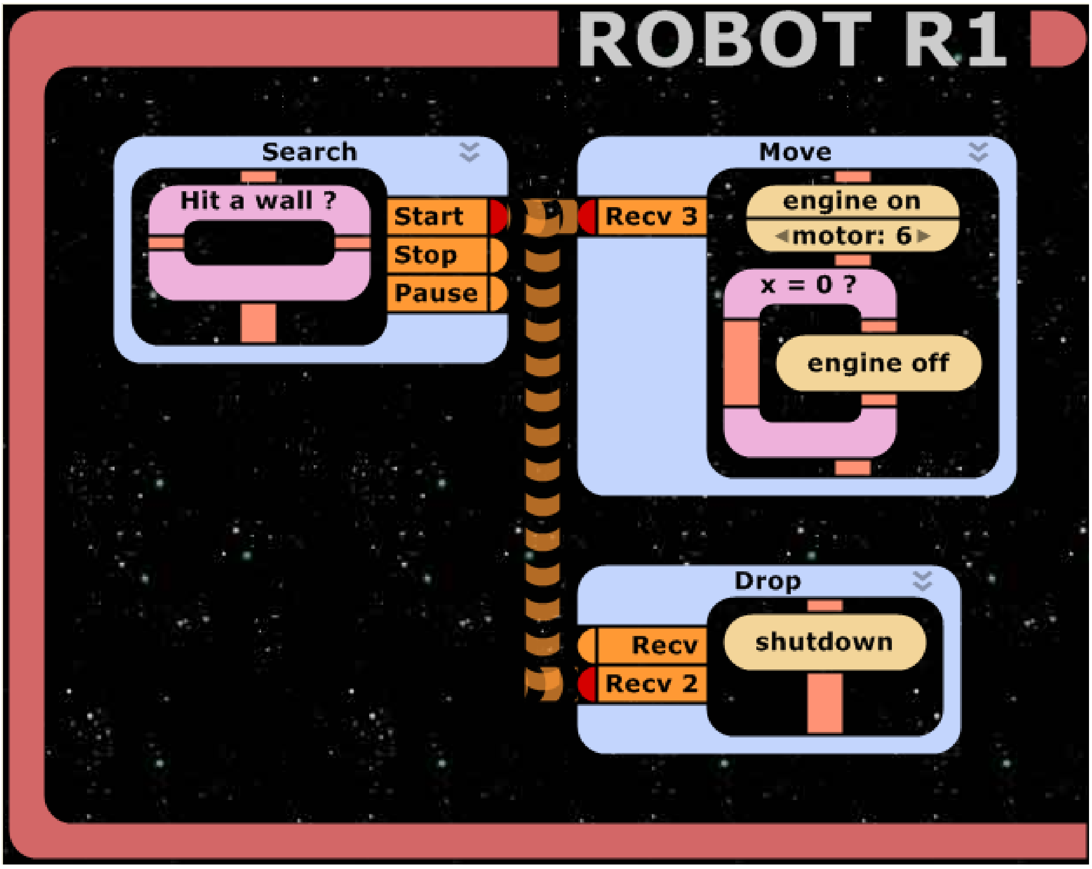


Figure 15.2: Design study of a new modeling language

language targets at pupils and students. Hence, its notation is closer to Star Trek than the simplistic UML notation.

Model-driven development is a very promising technology. It can combine the benefits of high level abstraction and efficient implementation. However, the tooling is essential and it is more complex than that of textual languages. The future will show whether UML will remain as the lingua franca of software modelers. Most likely, new application domains will bring up new modeling languages or derivations of existing ones.

Bibliography

[Aag01] Aagedal, J. Ø.: *Quality of Service Support in Development of Distributed Systems.* Oslo, Norway, University of Oslo, PhD., 2001

[AE02] Aagedal, J. Ø. ; Ecklund, E. F.: Modelling QoS: Towards a UML Profile. In: Jézéquel, J.-M. (Ed.) ; Hussmann, H. (Ed.) ; Cook, S. (Ed.): *UML 2002 - The Unified Modeling Language. Model Engineering, Concepts, and Tools. 5th International Conference, Dresden, Germany.* Heidelberg : Springer Verlag, October 2002. – ISBN 3–540–44254–5, p. 275–289

[AEH+99] Andries, M. ; Engels, G. ; Habel, A. ; Hoffmann, B. ; Kreowski, H. J. ; Kuske, S. ; Plump, D. ; Schuerr, A. ; Taentzer, G.: Graph Transformation for Specification and Programming. In: *Science of Computer Programming* 34 (1999), August, Nr. 1, p. 1–54

[Agi02] Agile Alliance. *Manifesto for Agile Software Development.* URL http://www.agilemanifesto.org/. November 2002

[Asp03] AspectJ Project. *AspectJ.* http://eclipse.org/aspectj. 2003

[ASU86] Aho, A. V. ; Sethi, R. ; Ullman, J. D.: *Compilers.* Addison-Wesley, January 1986. – ISBN 0–201–10088–6

[BBN03] BBN Technologies. *Quality Objects (QuO).* http://quo.bbn.com. November 2003

[Bec99] Beck, K.: *Extreme Programming Explained: Embrace Change.* Addison-Wesley, October 1999. – ISBN 0–201–61641–6

[Bec01] Becker, C.: *Dienstgüte-Management in verteilten Objektsystemen.* Frankfurt/Main, Germany, Universität Frankfurt/Main, PhD., 2001

[BG97] Becker, C. ; Geihs, K.: MAQS - Management for Adaptive QoS-enabled Services. In: *IEEE Workshop on Middleware for Distributed Real-Time Systems and Services, San Francisco, CA, USA*, 1997

[BG98] Becker, C. ; Geihs, K. *Quality of Service - Aspects of Distributed Programs.* International Workshop on Aspect-Oriented Programming at ICSE'98. 1998

[BG99] Becker, C. ; Geihs, K.: Generic QoS Specifications for CORBA. In: Steinmetz, R. (Ed.): *Kommunikation in Verteilten Systemen (KIVS'99), Darmstadt, Germany.* Heidelberg : Springer Verlag, March 1999 (Informatik aktuell). – ISBN 3–540–65597–2, p. 184–195

[BJPW99] Beugnard, A. ; Jézéquel, J.-M. ; Plouzeau, N. ; Watkins, D.: Making Components Contract Aware. In: *IEEE Computer* 13 (1999), July, Nr. 7, p. 38–45

[BKPPT01] Bottoni, P. ; Koch, M. ; Parisi-Presicce, F. ; Taentzer, G.: A Visualization of OCL Using Collaborations. In: Gogolla, M. (Ed.) ; Kobryn, C. (Ed.): *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada.* Heidelberg : Springer Verlag, October 2001 (LNCS 2185). – ISBN 3–540–42667–1, p. 257–271

[BKPPT02] Bottoni, P. ; Koch, M. ; Parisi-Presicce, F. ; Taentzer, G.: Working on OCL with Graph Transformation. In: *APPLIGRAPH Workshop on Applied Graph Transformation (AGT 2002), Grenoble, France*, 2002, p. 1–10

[Boh13] Bohr, N.: On the Constitution of Atoms and Molecules. In: *Philosophical Magazine* 26 (1913), July, p. 1–25

[Boo93] Booch, G.: *Object-Oriented Analysis and Design with Applications.* Addison-Wesley, October 1993. – ISBN 0–805–35340–2

[Bos04] Bosch, Hieronymus. *The Garden of Earthly Delights.* Prado, Madrid, URL http://www.ibiblio.org/wm/paint/auth/bosch/delight. 1504

[Box97] Box, D.: *Essential COM.* Addison-Wesley, December 1997. – ISBN 0–201–63446–5

[BR02] Born, M. ; Reznik, J. *UML Profile for CORBA Components.* OMG's Third Workshop on UML for Enterprise Applications: Model Driven Solutions for the Enterprise, Burlingame (USA). October 2002

[Bre02] Breker, T.: *Verwaltung und Austausch von UML-Modellen.* Frankfurt/Main, Germany, Universität Frankfurt/Main, Diplomarbeit, 2002

[CH03] Czarnecki, K. ; Helsen, S. *Classification of Model Transformation Approaches.* The Third OOPSLA Workshop on Domain-Specific Modeling, OOPSLA 2003 Workshop, Anaheim, CA, USA – http://www.softmetaware.com/oopsla2003/czarnecki.pdf. October 2003

[CHM+02] Csertán, G. ; Huszerl, G. ; Majzik, I. ; Pap, Z. ; Pataricza, A. ; Varró., D.: VIATRA - Visual Automated Transformations for Formal Verification of UML Models. In: *International Conference on Software Engineering (ASE 2002), Edinburgh, Scotland*, IEEE Computer Society, September 2002. – ISBN 0–7695–1736–6, p. 267–270

[Cla01] Clarke, Siobhán: *Composition of Object-Oriented Software Design Models.* Dublin, Ireland, Dublin City University, PhD., January 2001

[CVS03] CVS Team. *Concurrent Versions System.* http://www.cvshome.org. November 2003

[DHO01] Demuth, B. ; Hussmann, H. ; Obermaier, S. ; Whittle, J. (Ed.). *Experiments With XMI Based Transformations of Software Models.* WTUML: Workshop on Transformations in UML, ETAPS 2001 Satellite Event, Genova, Italy. April 2001

[Dou99] Douglass, P. B.: *Real-Time UML: Developing Efficient Objects for Embedded Systems.* 2nd edition. Addison-Wesley, October 1999. – ISBN 0201657848

[DW98] D'Souza, D. F. ; Wills, A. C.: *Objects, Components, and Frameworks with UML : The Catalysis(SM) Approach.* 1st edition. Addison-Wesley, October 1998. – ISBN 0201310120

[ECM01a] ECMA: C# Language Specification / European Computer Manufacturers Association. Geneva, Switzerland, 2001 (334). – ECMA Standard

[ECM01b] ECMA: Common Language Infrastructure / European Computer Manufacturers Association. Geneva, Switzerland, 2001 (335). – ECMA Standard

[EEKR99] Ehrig, H. ; Engels, G. ; Kreowski, H.-J. ; Rozenberg, G.: *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools.* World Scientific Pub Co, October 1999. – ISBN 9–810–24020–1

[FBB+99] Fowler, M. ; Beck, K. ; Brant, J. ; Opdyke, W. ; Roberts, D.: *Refactoring: Improving the Design of Existing Code.* 1st edition. Addison-Wesley, June 1999. – ISBN 0–201–48567–2

[Fis03] Fischertechnik. *Computing.* URL http://www.fischertechnik.com/html/computing.html. 2003

[FK98a] Frolund, S. ; Koistinen, J.: QML: A Language for Quality of Service Specification / HP Labs. 1998 (TR-98-10). – technical report

[FK98b] Frolund, S. ; Koistinen, J.: Quality of Service Specification in Distributed Object Systems Design. In: *4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS 98), Santa Fe, New Mexico*, 1998

[FK99] Frolund, S. ; Koistinen, J.: Quality-of-Service Aware Distributed Object Systems. In: *5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS 99), San Diego, CA, USA*, 1999

[FNTZ98] Fischer, T. ; Niere, J. ; Torunski, L. ; Zündorf, A.: Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In: Ehrig, H. (Ed.) ; Engels, G. (Ed.) ; Kreowski, H.-J. (Ed.) ; Rozenberg, G. (Ed.): *Theory and Application of Graph Transformations, 6th International Workshop (TAGT'98), Paderborn, Germany.* Heidelberg : Springer Verlag, November 1998 (LNCS 1764). – ISBN 3–540–67203–6, p. 296–309

[Fow97] Fowler, M.: *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* 2nd edition. Addison-Wesley, August 1997. – ISBN 0–201–32563–2

[Fow01] Fowler, M. *Is Design Dead?* published by SDMagazine, URL http://www.martinfowler.com/articles/designDead.html. February 2001

[Fri02] Friedl, J. E. F.: *Regular Expressions.* 2nd edition. O'Reilly, July 2002. – ISBN 0–596–00289–0

[GHS02] Gogolla, M. ; Henderson-Sellers, B.: Analysis of UML Stereotypes within the UML Metamodel. In: Jézéquel, J.-M. (Ed.) ; Hussmann, H. (Ed.) ; Cook, S. (Ed.): *UML 2002 - The Unified Modeling Language. Model Engineering, Concepts, and Tools. 5th International Conference, Dresden, Germany.* Heidelberg : Springer Verlag, October 2002. – ISBN 3–540–44254–5, p. 84–99

[GJ90] Garey, M. R. ; Johnson, D. S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman Company, November 1990. – ISBN 0–716–71045–5

[GLS01] Gill, C. D. ; Levine, D. L. ; Schmidt, D. C.: The Design and Performance of a Real-Time CORBA Scheduling Service. In: *Real-Time Systems* 20 (2001), Nr. 2, p. 117–154

[Gre01] Greenfield, J.: UML/EJB Mapping Specification 1.0 Public Review Draft / Rational Software Corporation. 2001. – Sun JSR-000026

[Gro01] Grosso, W.: *Java RMI.* 1st edition. O'Reilly & Associates, October 2001. – ISBN 1565924525

[GTK03] GTK Development Team. *GTK+ – The GIMP Toolkit.* http://www.gtk.org/. November 2003

[HBG+99] Hauck, F. J. ; Becker, U. ; Geier, M. ; Meier, E. ; Rastofer, U. ; Steckermeier, M.: The AspectIX Approach to Quality-of-Service Integration into CORBA / Friedrich-Alexander University Erlangen-Nürnberg, Germany. 1999 (TR-I4-99-09). – technical report

[HBG+01] Hauck, F. J. ; Becker, U. ; Geier, M. ; Meier, E. ; Rastofer, U. ; Steckermeier, M.: Aspectix: A Quality-Aware, Object-Based Middleware Architecture. In: Zielinski, K. (Ed.) ; Geihs, K. (Ed.) ; Laurentowski, A. (Ed.): *3rd International Working Conference on Distributed Applications and Interoperable Systems (DAIS'01), Kraków, Poland* Bd. 198, Kluwer, 2001. – ISBN 0–7923–7481–9, p. 115–120

[HE00] Heckel, R. ; Engels, G.: Graph Transformation and Visual Modeling Techniques. In: *BEATCS* 71 (2000), June, p. 186–203

[HEV03] Hejlsberg, A. ; Eckel, B. ; Venners, B. *The C# Design Process.* URL http://www.artima.com/intv/csdes.html. 2003

[HJGP99] Ho, W. M. ; Jézéquel, J.-M. ; Guennec, A. ; Pennaneac'h, F.: UMLAUT: An Extendible UML Transformation Framework. In: *14th International Conference on Automated Software Engineering (ASE 99), Florida, US*, IEEE Computer Society, 1999. – ISBN 0–7695–0415–9, p. 275–278

[HMU00] Hopcroft, J. E. ; Motwani, R. ; Ullman, J. D.: *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, November 2000. – ISBN 0–201–44124–1

[HPP00] Ho, W. M. ; Pennaneac'h, F. ; Plouzeau, N.: UMLAUT: A Framework for Weaving UML-Based Aspect-Oriented Designs. In: *Technology of Object-Oriented Languages and Systems (TOOLS 33), St. Malo, France*, IEEE Computer Society, June 2000, p. 324–335

[IET98] IETF Differentiated Services Working Group: An Architecture for Differentiated Services / IETF. 1998 (2475). – RFC

[Int03] Interactive Objects Software. *ArcStyler.* http://www.arcstyler.com. 2003

[Ise00] Iseminger, D.: *Com+ Developer's Reference Library.* Microsoft Press, September 2000. – ISBN 0–735–61138–6

[JBR99] Jacobsen, I. ; Booch, G. ; Rumbaugh, J.: *The unified software development process.* 1st edition. Addison-Wesley, January 1999. – ISBN 0–201–57169–2

[Jon01] Jones, S. P.: Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In: *Engineering theories of software construction* (2001), p. 47–96. ISBN 1–586–03172–4

[JPWG02] Jézéquel, J.-M. ; Plouzeau, N. ; Weis, T. ; Geihs, K. *From Contracts to Aspects in UML Designs.* Aspect-Oriented Modeling with UML, AOSD Workshop, Enschede, Netherlands. April 2002

[Kaf16] Kafka, Franz: *Die Verwandlung.* Leipzig, Germany : Kurt Wolff Verlag, 1916

[KCCB02] Kon, F. ; Costa, F. ; Campbell, R. ; Blair, G.: The Case for Reflective Middleware. In: *CACM* 45 (2002), June, Nr. 6, p. 33–38

[Ken03] Kennedy Carter. *eXecutable UML (xUML).*
URL http://www.kc.com/MDA/xuml.html. 2003

[KH02] Kovse, J. ; Harder, T.: Generic XMI-Based UML Model Transformations. In: Bruel, J.-M. (Ed.) ; Bellahsène, Z. (Ed.): *Int. Conf. on Object-Oriented Information Systems (OOIS'02), Montpellier, France.* Heidelberg : Springer Verlag, September 2002 (LNCS 2426), p. 192–198

[KLM+97] Kiczales, G. ; Lamping, J. ; Mendhekar, A. ; Maeda, C. ; Lopes, C. V. ; Loingtier, J.-M. ; Irwin, J.: Aspect-Oriented Programming. In: Aksit, M. (Ed.) ; Matsuoka, S. (Ed.): *11th European Conference on Object-Oriented Programming (ECOOP 97), Jyväskylä, Finnland.* Heidelberg : Springer Verlag, June 1997 (LNCS 1241). – ISBN 3–540–63089–9, p. 220–242

[Knu92] Knuth, D. E.: *Literate Programming.* CSLI Publications, May 1992. – ISBN 0–521–07380–6

[KTW02] Kiesner, C. ; Taentzer, G. ; Winkelmann, J.: A Visual Notation of the Object Constraint Language / TU-Berlin. 2002 (No. 2002/23). – technical report

[Kuh03] Kuhnen, Marcus: *Integration of Bandwidth Control Mechanisms in .NET Remoting*, TU-Berlin, Diplomarbeit, December 2003

[LB93] Löwe, M. ; Beyer, M.: AGG - An Implementation of Algebraic Graph Rewriting. In: *Rewriting Techniques and Applications, 5th International Conference (RTA-93), Montreal, Canada.* Heidelberg : Springer Verlag, June 1993 (LNCS 690). – ISBN 3–540–56868–9, p. 451–456

[LBS+98] Loyall, J. P. ; Bakken, D. E. ; Schantz, R. E. ; Zinky, J. A. ; Karr, D. A. ; Vanegas, R. ; Anderson, K. R.: QoS Aspect Languages and Their Runtime Integration. In: O'Hallaron, D. R. (Ed.): *4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98), Pittsburgh, USA.* Heidelberg : Springer Verlag, May 1998 (LNCS 1511). – ISBN 3–540–65172–1, p. 303–318

[Leg03] Lego. *Lego Mindstorms.* URL http://www.legomindstorms.com. 2003

[Mar02] Martin, J. C.: *Introduction to Languages and the Theory of Computation.* McGraw Hill, August 2002. – ISBN 0–071–19854–7

[Mey92] Meyer, B.: Applying Design by Contract. In: *IEEE Computer* 25 (1992), October, Nr. 10, p. 40–51

[Mey00] Meyer, B.: *Object-Oriented Software Construction.* 2nd edition. Prentice Hall, March 2000. – ISBN 0–136–29155–4

[Müh02] Mühl, Gero: *Large-Scale Content-Based Publish/Subscribe Systems.* Darmstadt, Germany, Darmstadt University of Technology, PhD., September 2002

[OMG02a] OMG: Common Object Request Broker Architecture: Core Specification / Object Management Group. 2002. – OMG formal document 02-12-06

[OMG02b] OMG: CORBA Components / Object Management Group. 2002. – OMG formal document 02-06-65

[OMG02c] OMG: Human-Usable Textual Notation (HUTN) Specification (ptc/02-12-01) / Object Management Group. 2002. – OMG final adopted specification 02-12-01

[OMG02d] OMG: Meta Object Facility / Object Management Group. 2002. – OMG formal document 02-04-03

[OMG02e] OMG: UML Profile for CORBA / Object Management Group. 2002. – OMG formal document 02-04-01

[OMG02f] OMG: UML Profile for Enterprise Distributed Object Computing Specification (ptc/02-02-05) / Object Management Group. 2002. – OMG adopted specification 02-02-05

[OMG03a] OMG. *Model Driven Architecture.* URL http://www.omg.org/mda. 2003

[OMG03b] OMG. *Object Management Group Home Page.* URL http://www.omg.org. 2003

[OMG03c] OMG: RealTime-CORBA / Object Management Group. 2003. – OMG formal document 03-11-01

[OMG03d] OMG: Response to the UML 2.0 OCL RfP (ad/2000-09-03), Version 1.6 / Object Management Group. 2003. – OMG revised submission 03-01-07

[OMG03e] OMG: UML 2.0 Infrastructure Specification / Object Management Group. 2003. – OMG formal document 03-09-15

[OMG03f] OMG: UML 2.0 Superstructure Specification / Object Management Group. 2003. – OMG formal document 03-08-02

[OMG03g] OMG: UML Profile for Schedulability, Performance, and Time Specification / Object Management Group. 2003. – OMG formal document 03-09-01

[OMG03h] OMG: Unified Modeling Language: Diagram Interchange (ptc/03-07-03), Version 2.0 / Object Management Group. 2003. – OMG draft adopted specification 03-07-03

[OMG03i] OMG: Unified Modeling Language (UML), version 1.5 / Object Management Group. 2003. – OMG formal document 03-03-01

[OMG03j] OMG: XML Metadata Interchange (XMI) Specification / Object Management Group. 2003. – OMG formal document 03-05-02

[PBG01] Peltier, M. ; Bezivin, J. ; Guillaume, G. ; Whittle, J. (Ed.). *MTRANS, a general framework based on XSLT, for model transformations.* WTUML: Workshop on Transformations in UML, ETAPS 2001 Satellite Event, Genova, Italy. April 2001

[PC03] Peterson, J. ; Chitil, O. *Haskell - a purely functional language.* http://www.haskell.org/. September 2003

[Per03] Perforce Software Inc. *Perforce.* http://www.perforce.com. November 2003

[Pet81] Peterson, J. L.: *Petri Net Theory and the Modeling of Systems.* 1st edition. Prentice Hall, June 1981. – ISBN 0136619835

[PLS$^+$00a] Pal, P. ; Loyall, J. ; Schantz, R. ; Zinky, J. ; Shapiro, R. ; Megquier, J.: Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration. In: *3rd International Symposium on Object-Oriented Real-time Distributed Computing (ISORC 2000), Newport Beach, CA, USA*, IEEE Computer Society, March 2000. – ISBN 0–7695–0607–0

[PLS$^+$00b] Pal, P. P. ; Loyall, J. P. ; Schantz, R. E. ; Zinky, J. A. ; Shapiro, R. ; Megquier, J.: Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration. In: *3rd IEEE International Symposium on Object-Oriented Real-time distributed Computing (ISORC 2000), Newport Beach, CA, USA*, IEEE Computer Society, March 2000. – ISBN 0–7695–0607–0, p. 310–319

[Plu93] Chapter 15 in: Plump, Detlef: *Term Graph Rewriting: Theory and Practice.* John Wiley & Sons, 1993. – ISBN 0–471–93567–0

[Plu98] Plump, Detlef: Termination of Graph Rewriting is Undecidable. In: *Fundamenta Informaticae* 33 (1998), February, Nr. 2, p. 201–209

[Pri57] Prim, R. C.: Shortest Connection Networks and Some Generalizations. In: *Bell System Tech. J.* (1957), Nr. 36, p. 1389–1401

[PSC03] Pyaraliy, I. ; Schmidt, D. C. ; Cytron, R. K.: Techniques for Enhancing Real-time CORBA Quality of Service. In: *Proceedings of the IEEE* 91 (2003), July, Nr. 7, p. 1070– 1085

[Pyt03] Python. *The Python Homepage.* http://www.python.org. November 2003

[PZB00] Peltier, M. ; Ziserman, F. ; Bezivin, J. *On levels of model transformation.* XML Europe 2000, Paris, France – http://www.gca.org/papers/xmleurope2000/papers/s36-02.html. June 2000

[RAC$^+$02] Robinson, S. ; Allen, S. ; Cornes, O. ; Glynn, J. ; Greenvoss, Z. ; Harvey, B. ; Nagel, C. ; Skinner, M. ; Watson, K.: *Professional C#.* 2nd edition. Wrox Press Ltd, May 2002. – ISBN 1–861007–04–3

[Ram02] Rammer, I.: *Advanced .NET Remoting.* 1st edition. APress, April 2002. – ISBN 1590590252

[RBC$^+$03] Ren, Y. ; Bakken, D. E. ; Courtney, T. ; Cukier, M. ; Karr, D. A. ; Rubel, P. ; Sabnis, C. ; Sanders, W. H. ; Schantz, R. E. ; Seri, M.: AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects. In: *IEEE Transactions on Computers* 52 (2003), January, Nr. 1, p. 31–50

[RBUW03] Ritter, T. ; Born, M. ; Unterschütz, T. ; Weis, T.: A QoS Metamodel and its Realization in a CORBA Component Infrastructure. In: *36th Hawaii International Conference on System Sciences (HICSS-36 2003), Big Island, HI, USA*, IEEE Computer Society, January 2003. – ISBN 0–7695–1874–5, p. 318

[Rit03] Ritter, T. *Qedo QoS-enabled distributed objects.* http://qedo.berlios.de/. November 2003

[RKR$^+$00] Reggio, G. ; Knapp, A. ; Rumpe, B. ; Selic, B. ; Wieringa, R. *Dynamic Behaviour in UML Models: Semantic Questions, Workshop at UML 2000, York, UK.* URL http://www.disi.unige.it/person/ReggioG/UMLWORKSHOP/PROGRAM.html. 2000

[RP03] Römer, K. ; Puder, A. *MICO is CORBA.* http://www.mico.org. October 2003

[Rut11] Rutherford, E.: The Scattering of a and b Particles by Matter and the Structure of the Atom. In: *Philosophical Magazine* 21 (1911), May, p. 669–688

[RZ03] Röttger, S. ; Zschaler, S.: CQML+: Enhancements to CQML. In: *1st International Workshop on Quality of Service in Component-Based Software Engineering, Toulouse, France*, Cépaduès-Éditions, June 2003. – ISBN 2–85428–617–0, p. 43–56

[SAD$^+$02] Sassen, A. ; Amoros, G. ; Donth, P. ; Geihs, K. ; Jezequel, J. ; Odent, K. ; Plouzeau, N. ; Weis, T. *QCCS: A Methodology for the Development of Contract-Aware Components Based on Aspect-Oriented Design.* Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, AOSD Workshop, Enschede, Netherlands. April 2002

[Sar02] Sarkar, S.: Model-Driven Programming using XSLT. In: *XML Journal* 3 (2002), August, Nr. 8, p. 42–51

[Sch99] Schnittger, G. *Effiziente Algorithmen.* Lecture at Universität Frankfurt/Main. 1999

[Sch00a] Schmidt, D.: *Patterns for Concurrent and Networked Objects.* 1st edition. John Wiley & Sons, September 2000. – ISBN 0471606952

[Sch00b] Schneier, B. *Software Complexity and Security.* URL http://www.counterpane.com/crypto-gram-0003.html. March 2000

[Sch03] Schmidt, D. C. *Real-time CORBA with TAO.* http://www.cs.wustl.edu/ schmidt/TAO.html. November 2003

[SPH$^+$01] Sunyé, G. ; Pennaneac'h, F. ; Ho, W. M. ; Guennec, A. ; Jézéquel, J.-M.: Using UML Action Semantics for executable modeling and beyond. In: Dittrich, K. R. (Ed.) ; Geppert, A. (Ed.) ; Norrie, M. C. (Ed.): *Advanced Information Systems Engineering (CAiSE 2001), Interlaken, Switzerland.* Heidelberg : Springer Verlag, June 2001 (LNCS 2068). – ISBN 3–540–42215–3, p. 433–447

[SPTJ01] Sunyé, G. ; Pollet, D. ; Traon, Y. ; Jézéquel, J. M.: Refactoring UML Models. In: Gogolla, M. (Ed.) ; Kobryn, C. (Ed.): *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada.* Heidelberg : Springer Verlag, October 2001 (LNCS 2185). – ISBN 3–540–42667–1, p. 134–148

[ST85] Sleator, D. D. ; Tarjan, R. E.: Self-adjusting binary search trees. In: *J. ACM* 32 (1985), Nr. 3, p. 652–686. – ISSN 0004–5411

[Str00] Stroustrup, Bjarne: *The C++ Programming Language.* Addison-Wesley, February 2000. – ISBN 0–201–70073–5

[Sub03] Subversion Team. *Subversion.* http://subversion.tigris.org. November 2003

[Sun03] Sun Microsystems. *Java Server Pages.* http://java.sun.com/products/jsp/. November 2003

[Szy02] Szyperski, C.: *Component Software.* Addison-Wesley, November 2002. – ISBN 0–201–74572–0

[Tae99] Taentzer, G.: AGG: A Tool Environment for Algebraic Graph Transformation. In: Nagl, M. (Ed.) ; Schürr, A. (Ed.) ; Münch, M. (Ed.): *Applications of Graph Transformations with Industrial Relevance, International Workshop, AGTIVE'99, Kerkrade, The Netherlands.* Heidelberg : Springer Verlag, September 1999 (LNCS 1779). – ISBN 3–540–67658–9, p. 481–488

[TFS03] TFS Theoretische Informatik / Formale Spezifikation, TU-Berlin. *The AGG Homebase.* http://tfs.cs.tu-berlin.de/agg/. April 2003

[The03] The precise UML group. *pUML.* http://www.puml.org. 2003

[Tip02] Tipler, P. A.: *Physik.* Spektrum Verlag, October 2002. – ISBN 3–486–25564–9

[TPV03] THEELEN, B. D. ; VAN DER PUTTEN, P. H. A. ; VOETEN, J. P. M.: Using the SHE Method for UML-based Performance Modelling. In: *System Specification and Design Languages Best of FDL'02, Gieres, France*. Boston : Kluwer Academic Publishers, April 2003. – ISBN 1–4020–7414–X

[Tri01] TRISKEL PROJECT, IRISA. *UMLAUT: Unified Modeling Language All pUrposes Transformer*. URL http://www.irisa.fr/UMLAUT. 2001

[Tro03] TROLLTECH. *Qt Overview*. http://www.trolltech.com/products/qt/. November 2003

[UWG03] ULBRICH, A. ; WEIS, T. ; GEIHS, K.: QoS Mechanism Composition at Design-Time and Runtime. In: *23rd International Conference on Distributed Computing Systems Workshops (ICDCS 2003 Workshops), Providence, RI, USA*, IEEE Computer Society, 2003. – ISBN 0–7695–1921–0, p. 118–124

[UWGB03] ULBRICH, A. ; WEIS, T. ; GEIHS, K. ; BECKER, C.: DotQoS - A QoS Extension for .NET Remoting. In: JEFFAY, K. (Ed.) ; STOICA, I. (Ed.) ; WEHRLE, K. (Ed.): *11th International Workshop on Quality of Service (IWQoS 2003), Berkeley, CA, USA*. Heidelberg : Springer Verlag, June 2003 (LNCS 2707). – ISBN 3–540–40281–0, p. 363–380

[VGP01] VARRÓ, D. ; GYAPAY, S. ; PATARICZA, A. ; WHITTLE, J. (Ed.). *Automatic Transformation of UML Models for System Verification*. WTUML: Workshop on Transformations in UML, ETAPS 2001 Satellite Event, Genova, Italy. April 2001

[VVP02] VARRÓ, D. ; VARRÓ, G. ; PATARICZA, A.: Designing the Automatic Transformation of Visual Languages. In: *Science of Computer Programming* 44 (2002), August, Nr. 2, p. 205–227

[VZL+98] VANEGAS, R. ; ZINKY, J. A. ; LOYALL, J. P. ; KARR, D. ; SCHANTZ, R. E. ; BAKKEN, D. E.: QuO's Runtime Support for Quality of Service in Distributed Objects. In: DAVIES, N. (Ed.) ; RAYMOND, K. (Ed.) ; SEITZ, J. (Ed.): *International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), The Lake Distruct, UK*. Heidelberg : Springer Verlag, September 1998. – ISBN 1–85–233–088–0, p. 207–223

[W3C98] W3C: Document Object Model (DOM) Level 1 Specification / W3C. 1998. – W3C recommendation http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/

[W3C99a] W3C: XML Path Language (XPath) Version 1.0 / W3C. 1999. – W3C recommendation http://www.w3.org/TR/xpath

[W3C99b] W3C: XSL Transformations (XSLT) Version 1.0 / W3C. 1999. – W3C recommendation http://www.w3.org/TR/xslt

[W3C01] W3C: XML Linking Language (XLink) Version 1.0 / W3C. 2001. – W3C recommendation http://www.w3.org/TR/xlink

[W3C02] W3C: XML Pointer Language (XPointer) / W3C. 2002. – W3C recommendation http://www.w3.org/TR/xptr

[WBGP01] Weis, T. ; Becker, C. ; Geihs, K. ; Plouzeau, N.: A UML Meta-model for Contract Aware Components. In: Gogolla, M. (Ed.) ; Kobryn, C. (Ed.): *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada.* Heidelberg : Springer Verlag, October 2001 (LNCS 2185). – ISBN 3–540–42667–1, p. 442–456

[WG00] Weis, T. ; Geihs, K.: Components on the Desktop. In: Mitchell, R. (Ed.) ; Jézéquel, J.-M. (Ed.) ; Bosch, J. (Ed.) ; Meyer, B. (Ed.) ; Wills, A. C. (Ed.) ; Woodman, M. (Ed.): *Technology of Object-Oriented Languages and Systems (TOOLS'33), Mont-Saint-Michel, France*, IEEE Computer Society, 2000. – ISBN 0–7695–0731–X, p. 250–261

[Whe01] Wheeler, D. *Estimating Linux's Size.* URL http://www.dwheeler.com/sloc. May 2001

[WPA$^+$03] Chapter 9 in: Weis, T. ; Plouzeau, N. ; Amoros, G. ; Donth, P. ; Geihs, K. ; Jézéquel, J.-M. ; Sassen, A.-M.: *Business Component-Based Software Engineering.* Kluwer Academic Publishers, 2003. – ISBN 1–4020–7207–4

[WSKP01] Wang, N. ; Schmidt, D. C. ; Kircher, M. ; Parameswaran, K.: Adaptive and Reflective Middleware for QoS-Enabled CCM Applications. In: *Distributed Systems online (dsonline.computer.org)* 2 (2001), Nr. 5

[WUG03a] Weis, T. ; Ulbrich, A. ; Geihs, K.: Model Metamorphosis. In: *IEEE Software* 20 (2003), September/October, Nr. 5, p. 46–51

[WUG03b] Weis, T. ; Ulbrich, A. ; Geihs, K.: Modellierung und Zusicherung nicht-funktionaler Eigenschaften bei Entwurf, Implementierung und zur Laufzeit verteilter Anwendungen. In: Irmscher, K. (Ed.) ; Fähnrich, K. (Ed.): *Kommunikation in Verteilten Systemen (KiVS).* Heidelberg : Springer Verlag, 2003 (Informatik Aktuell). – in German. – ISBN 3–540–00365–7

[WUG03c] Weis, T. ; Ulbrich, A. ; Geihs, K. *Quality of Service Engineering with UML and MDA.* Tutorial presented at NetObjectDays'03, Erfurt, Germany. September 2003

[WUG03d] Weis, T. ; Ulbrich, A. ; Geihs, K.: Quality of Service Engineering with UML, .NET, and CORBA. In: *25th International Conference on Software Engineering (ICSE 2003), Portland, Oregon, USA*, IEEE Computer Society, 2003. – half-day tutorial. – ISBN 0–7695–1877–X, p. 759–761

[ZBS97a] Zinky, J. A. ; Bakken, D. E. ; Schantz, R. E.: Architectural support for quality of service for CORBA objects. In: *In Theory and Practice of Object Systems* 3 (1997), Nr. 1, p. 55–73

[ZBS97b] Zinky, J. A. ; Bakken, D. E. ; Schantz, R. E.: Architectural Support for Quality of Service for CORBA Objects. In: *Theory and Practice of Object Systems* 3 (1997), Nr. 1, p. 55–73

All web references have been valid in November 2003. If the referenced web document has a publishing date, this date is shown in the bibliography entry. Otherwise, November 2003 is used as date of publishing.

Appendix A

List of Symbols

$\mathbb{N}$	is the set of natural numbers
$\mathbb{B}$	is the set $\{true, false\}$
$G = (V, E)$	G is a tree with vertices V and edges E
e	is an edge
v	is a vertex
G_{span}	is the spanning tree of G
r	is a transformation rule
L	is the left-hand side of a transformation rule
R	is the right-hand side of a transformation rule
K	is the interface graph of a transformation rule
$appl$	is the application condition of a transformation rule
map_{XY}	maps vertices of graph X to vertices of graph Y
$G_1 \Rightarrow_r G_2$	is a direct derivation
ℓ_X	is a labeling function for set X
$\mathcal{M}$	is a meta model
$V_{\mathcal{M}}$	are the meta classes of $\mathcal{M}$
$E_{\mathcal{M}}$	are the meta associations of $\mathcal{M}$
$a_{\mathcal{M}(v)}$	is the set of meta attributes of meta class v
$top(V_{\mathcal{M}})$	are the top-level (i.e. root) meta classes of $\mathcal{M}$
M	is a model
$t_v(x)$	is the meta class of a model element x
$\mathcal{G}$	is the PIM meta model
$\mathcal{S}$	is the PSM meta model
$\mathcal{R}$	is the mapping meta model
$\mathcal{H}$	is the merged meta model
I_n	is the n-th version of a PIM
S_n	is the n-th version of a PSM
$\tilde{S}_n$	is the user modified version of S_n
$ID(x)$	is a unique ID for model element x
T	is a Kafka transformation
f	is either the final state vertex of T or an evaluation function
m_i	is a measurand
$[m_i]$	is a sequence of metered values of m_i
M_i	is the range of m_i
$[M_i]$	is the set of all possible sequences $[m_i]$
p_i	is a parameter of an evaluation function
P	is the range

Appendix B

Meta Models

B.1 DotQoS Meta Model

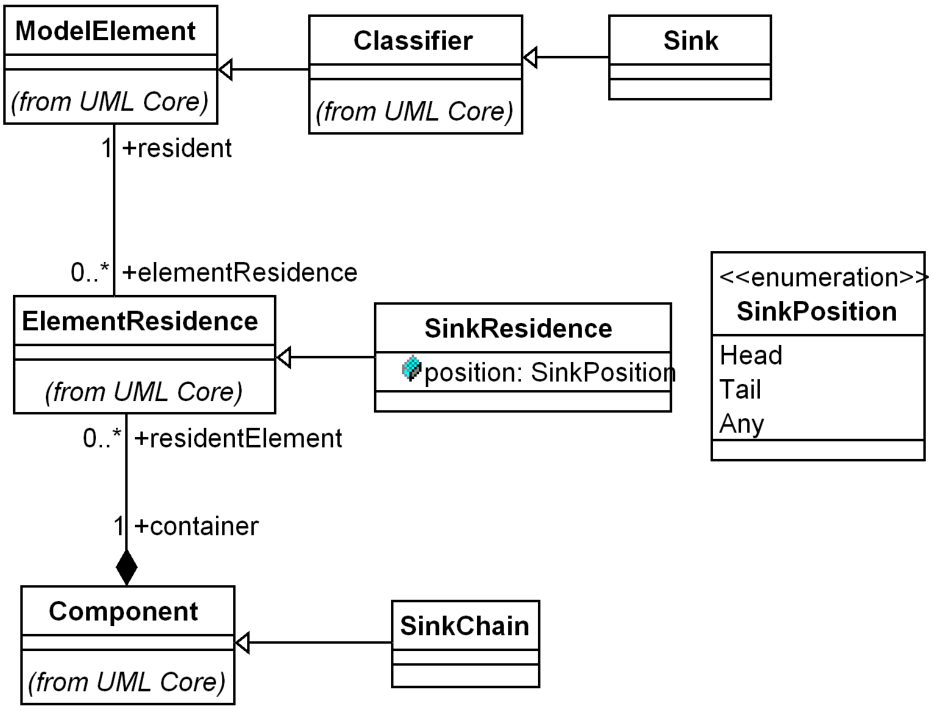


Figure B.1: Meta model extension for DotQoS

B.2 XML Meta Model

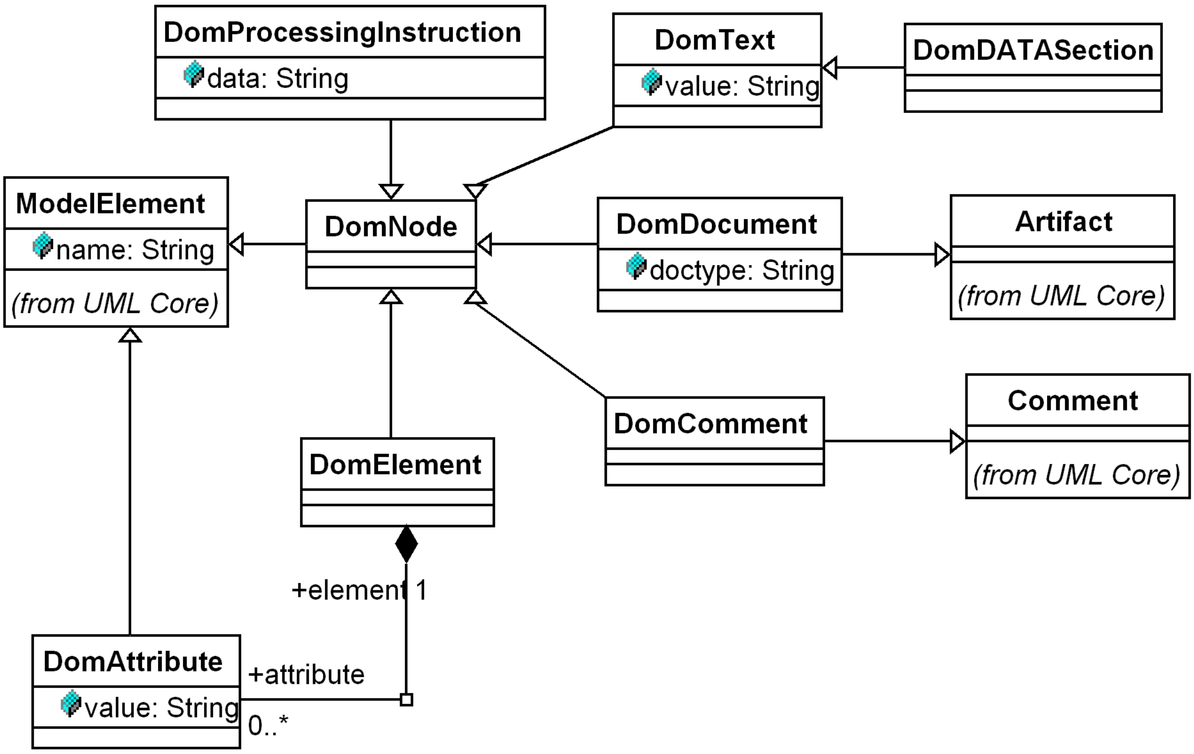


Figure B.2: Meta model extension for XML

B.3 QoS Meta Model

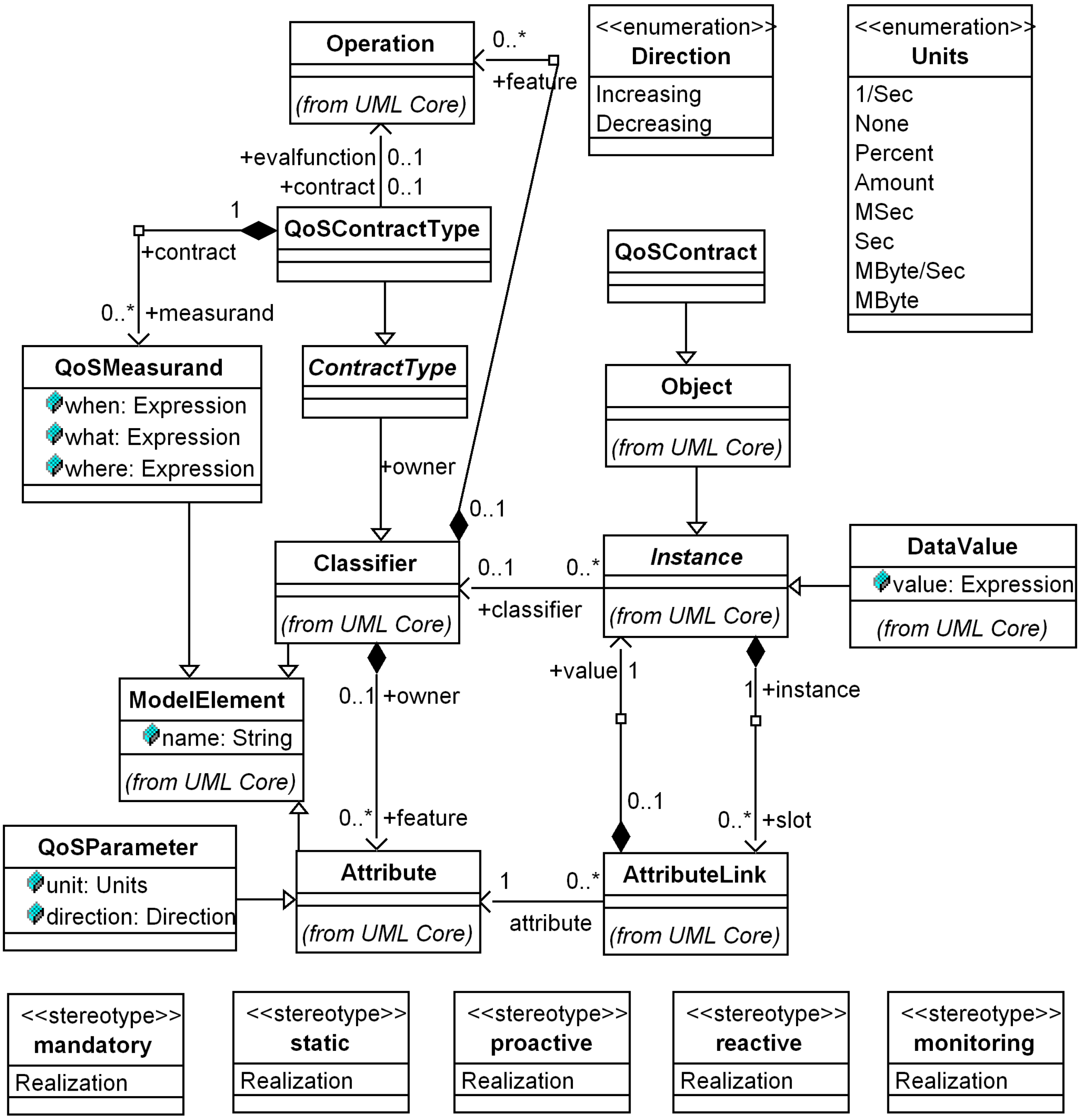


Figure B.3: Meta model extension for QoS

Lebenslauf

Name:	Torben Weis
29.11.1973	Geboren in Frankfurt am Main
1980 - 1984	Besuch der Grundschule in Bad Vilbel
1984 - 1986	Besuch der J.F.Kennedy Schule in Bad Vilbel
1986 - 1993	Besuch des Büchner Gymnasiums in Bad Vilbel
1993	Abitur
1994 - 2000	Studium der Informatik an der J.W.Goethe Universität Frankfurt am Main mit Abschluss Diplom Informatiker
Jan. 2001 - Sep. 2001	Wissenschaftlicher Mitarbeiter an der J.W.Goethe Universität Frankfurt am Main
Seit Sep. 2001	Wissenschaftlicher Mitarbeiter an der TU Berlin

Selbständigkeitserklärung

Hiermit erkläre ich, die vorliegende Arbeit selbständig ohne fremde Hilfe verfaßt und nur die angegebene Literatur und Hilfsmittel verwendet zu haben.

Torben Weis
26. Januar 2004